

Integration of the Process Algebra CSP in Dependent Type Theory - Formalisation and Verification

Bashar Igried Deb Alkhawaldeh

April 26, 2018

Integration of the Process Algebra CSP in Dependent Type Theory - Formalisation and Verification

Bashar Igried Deb Alkhawaldeh
PhD Thesis

Submitted to Swansea University in fulfilment of the requirements for the
Degree of Doctor of Philosophy

April 26, 2018

Swansea University



Prifysgol Abertawe
Swansea University

Department of Computer Science

Examiners **Conor McBride**
Computer and Information Sciences
The University of Strathclyde

John Tucker
Department of Computer Science
Swansea University

Supervisors **Anton Setzer**
Ulrich Berger

Bashar Igried Deb Alkhawaldeh

Integration of the Process Algebra CSP in Dependent Type Theory - Formalisation and Verification

PhD Thesis, April 26, 2018

Examiners: Conor McBride

John Tucker

Supervisor: Anton Setzer

Ulrich Berger

Swansea University

Department of Computer Science

Faraday Tower, Swansea University

Swansea, SA2 8PP

United Kingdom

Declaration

I declare that the work presented in this thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Swansea, April 26, 2018

Bashar Igried Deb
Alkhawaldeh

I declare that this thesis is the result of my own investigations, except where otherwise stated and that other sources are acknowledged by footnotes giving explicit references and that a bibliography is appended.

Swansea, April 26, 2018

Bashar Igried Deb
Alkhawaldeh

I declare that I give consent for the thesis, if accepted, to be made available online in the University's Open Access Repository and for inter-library loan, and for the title and summary to be made available to outside organisations.

Swansea, April 26, 2018

Bashar Igried Deb
Alkhawaldeh



Contents

1	Introduction	5
1.1	Motivation	5
1.2	CSP	5
1.3	The IO Monad in Agda	6
1.4	Monadic Processes	7
1.5	Proofs of Correctness of CSP-processes in Agda	8
1.6	Main Achievements	9
1.7	Thesis Outline	10
1.8	Published Material	12
1.8.1	Refereed Publications	12
1.8.2	Refereed Short Papers	13
1.8.3	Papers in Preparation	14
1.8.4	Conferences, Workshop & talk	15
2	CSP	17
2.1	Communicating Sequential Processes	18
2.2	Framework of CSP	19
2.2.1	Transitions	19
2.2.2	Inference rules	19
2.2.3	Events	19
2.3	The Syntax of CSP	21
2.3.1	Primitive Processes	21
2.3.2	Sequential Composition	22
2.3.3	Event Prefix	23
2.3.4	Recursion	23
2.3.5	Internal Choice	24
2.3.6	External Choice	24
2.3.7	Parallel and Interleaving	25

2.3.8	Hiding	27
2.3.9	Renaming	28
2.3.10	Interrupt	29
2.4	The Semantics of CSP	29
2.4.1	Trace Semantics	29
2.4.2	Stable Failures Semantics	30
2.4.3	Failures, Divergences, and Infinite Traces Semantics	31
2.4.4	Semantics of Recursive Processes	32
2.4.5	Bisimilarity for CSP	32
2.5	Tool Support	36
2.5.1	ProBE	36
2.5.2	Failures Divergences Refinement (FDR)	37
2.5.3	CSP-Prover	38
3	Theorem Prover Agda	39
3.1	Totality of Agda	41
3.1.1	Type checking	41
3.1.2	Coverage checking	42
3.1.3	Termination checking	43
3.1.4	Equality	45
3.2	Types and Expressions In Agda	45
3.2.1	Inductive Data Types	46
3.2.2	General Form of Inductive Data Types	47
3.2.3	Dependent function type	47
3.2.4	Guardedness checking in coinductive programs	48
3.2.5	Coinductive Data Types	48
3.2.6	Induction-Recursion	51
3.2.7	Records	53
3.2.8	Mixfix Operators and Unicode	55
3.2.9	Implicit Arguments	56
3.2.10	Module System	57
3.2.11	Postulated Types	58
3.2.12	Let, Where, With-expressions, Mutual Definitions, Intensional Equality, and Rewrite	58
3.2.13	BUILTIN and Primitive	61
3.3	Setup of Agda	62
3.3.1	Interactive Interface	62
3.3.2	Compiled Version of Agda	62
3.3.3	Interactive Programs in Agda	63

4	Review of Literature	67
4.1	Formal methods	67
4.2	Process algebra	69
4.2.1	CSP	69
4.3	Theorem Provers	70
4.4	Defining Process Algebras in Functional Programming . . .	71
4.5	Defining Process Algebras in Dependent Type Theory . . .	73
4.6	Defining Process Algebras using Coalgebras	74
4.7	Agda as Platform for Modelling Programs	75
5	The Library CSP-Agda	77
5.1	Label Universe LUniv	77
5.2	Monadic Composition in CSP-Agda	80
5.3	Representing CSP Processes in Agda	83
5.4	Three Kinds of Termination of Processes	88
5.5	Sequential Composition	88
5.6	The Recursion Operator	90
5.7	STOP, SKIP, Terminate, DIV	92
5.8	Prefix	93
5.9	Internal Choice	94
5.10	External Choice	95
5.11	Renaming	101
5.12	Hiding	102
5.13	Interleaving Operator	103
5.14	The Parallel Operator	105
5.15	Interrupt Operator	109
6	A Simulator for CSP-Agda	113
7	Trace Semantics for CSP-Agda	121
7.1	Proof of the Algebraic Laws	124
7.2	Proof of the Laws of Refinement	124
7.3	Proof of the Monadic Laws	125
7.4	Proof of Commutativity of the Interleaving Operator . . .	129
7.5	Proof of Commutativity of the Parallel Operator	130
7.6	Proof of Commutativity of the External Choice Operator .	133
7.7	Proof of Associativity of the Interleaving Operator	135
7.8	Proof of Associativity of the External Choice Operator . .	136
7.9	Proof of Law for Renaming Operator	138
7.10	Proof of Laws for the Hiding Operator	139
7.11	Proof of Termination and Unit for Interleaving Operator .	140

7.12	Proof of the Law of the Parallel Operator combined with terminate	142
8	Stable Failures Semantics	145
8.1	Traces with Next process	146
8.2	Stable Process	148
8.3	Refusal Sets	150
8.4	Stable Failures	152
8.5	Refinement Relations	153
8.6	Proofs for Stable Failures Semantics	154
9	Failures Divergences Infinite Traces Semantics	155
9.1	Motivation	155
9.2	Failures	156
9.3	Divergent Process	156
9.4	Divergent Traces	157
9.5	Infinite Traces	158
9.6	Refinement Relations	159
9.7	Proofs in Failures Divergences Infinite Traces Semantics . .	161
10	Bisimulation	163
10.1	Defining Strong Bisimilarity for CSP-Agda	164
10.2	Defining Divergence-Respecting Weak Bisimilarity for CSP-Agda	166
10.3	Proof of Reflexivity for Strong Bisimilarity	172
10.4	Proof of Reflexivity for Divergence-Respecting Weak Bisim- ilarity	173
10.5	Proof of Symmetry for Divergence-Respecting Weak Bisim- ilarity	174
10.6	Proof that Strong Bisimilarity Implies Divergence-Respecting Weak Bisimilarity	176
10.7	Bisimilarity Implies Trace Equivalence	178
10.7.1	Strong Bisimilarity Implies Trace Equivalence	178
10.7.2	Divergence-Respecting Weak Bisimilarity Implies Trace Equivalence	180
10.8	Proof of Lemma 2.4.6 Part 1	186
10.9	Proof of Lemma 2.4.6, Part 2 (Obtaining Roscoe Stability) .	200
10.10	Bisimilarity Implies Stable Failures Equivalence	205
10.11	Bisimilarity Implies Failures Divergences Infinite Equivalence	209
10.11.1	DRW-Bisimilarity Implies Refinement with respect to Failures	209

10.11.2	DRW-Bisimilarity Implies Refinement with respect to Divergences	212
10.11.3	DRW-Bisimilarity Implies Refinement with respect to Infinite Traces	214
10.11.4	DRW-Bisimilarity Implies Refinement with respect to FDI	217
10.12	Proofs in Divergence-Respecting Weak Bisimilarity Semantics	219
10.12.1	Proof of Commutativity of the External Choice Operator	219
10.12.2	Proof of Commutativity of the Interleaving Operator	222
10.13	Proof of the Monadic Laws	224
10.13.1	Proof of First Monadic Law	224
10.13.2	Proof of Third Monadic Law	226
10.13.3	The Second Monadic Law	229
11	Railway Interlocking Systems	233
11.1	Specifying an Possible Scenario for Railways in Natural Language	234
11.2	CSP and CSP-Agda Specification	234
11.3	Verification of a Possible Scenario for Railways Using FDR .	241
11.4	Verification/Simulation of the Case Study	243
11.4.1	Simulate using ProBE	244
11.4.2	Simulation using CSP-Agda-Simulator	244
11.4.3	Verification in CSP-Agda	245
11.5	Correcting the System	245
12	Summary	251
13	Future Work	253
	Bibliography	254
	Appendices	277
A	Agda Code	279
A.1	addTick.agda	279
A.2	addTickOperator.agda	282
A.3	auxData.agda	284
A.4	auxLemmaPar.agda	285
A.5	bisimForNextProcess.agda	287
A.6	bisimilarity.agda	304
A.7	bisimilarityProofs.agda	310

A.8	bisimilarityProofsWithSchneiderStable2.agda	336
A.9	bisimilarityProofsWithSchneiderStable3.agda	363
A.10	bisimilarityProofsWithSchneiderStable3Part2.agda	383
A.11	bisimilarityProofsWithSchneiderStable3Part2TheoremOnly.agda	391
A.12	bisimilarityProofsWithSchneiderStable3Part2WeakerVersion.agda	397
A.13	bisimilarityProofsWithSchneiderStable3Part3.agda	403
A.14	bisimilarityProofsWithSchneiderStable.agda	410
A.15	bisimImpliesBisim.agda	436
A.16	bisimImpliesFDI.agda	438
A.17	bisimImpliesFDIPartTwo.agda	441
A.18	bisimImpliesRefinementInfiniteTraces.agda	450
A.19	bisimImpliesTraceEquiv.agda	451
A.20	bisimLemFmap.agda	460
A.21	bisimSImpliesBisimw.agda	464
A.22	bisimSym.agda	467
A.23	bisimwImpliesStableFailuresEquivalence.agda	470
A.24	choiceAuxFunction.agda	479
A.25	choiceFromList.agda	481
A.26	choiceSetU.agda	482
A.27	choiceSetUOptimized.agda	485
A.28	choiceSetUOptimized2.agda	487
A.29	choiceSetUOptimized3.agda	488
A.30	dataAuxFunction.agda	489
A.31	div.agda	493
A.32	efq.agda	494
A.33	example.agda	494
A.34	example2.agda	499
A.35	example3.agda	501
A.36	externalChoice.agda	502
A.37	fdi.agda	512
A.38	fdiImpliesEquivalence.agda	520
A.39	fdiOld.agda	521
A.40	fdiPart2.agda	521
A.41	fdiRefusal.agda	527
A.42	fdiRefusalPartsRemoved.agda	542
A.43	hidingOperator.agda	543
A.44	interleave.agda	546
A.45	internalChoice.agda	552
A.46	interrupt.agda	554
A.47	IOExampleScreenShotForTyDePaper2.agda	557
A.48	IOSeqCom.agda	558

A.49	label.agda	559
A.50	labelEq.agda	560
A.51	labelUniv.agda	560
A.52	labelUnivAsPureRecord.agda	563
A.53	labelUnivAsUniverse.agda	565
A.54	lemFmap.agda	567
A.55	libBool.agda	572
A.56	libEq.agda	573
A.57	libList.agda	576
A.58	maybe.agda	576
A.59	monadicbind.agda	576
A.60	monadicbindFixing2ndMonadLaw.agda	578
A.61	NativeIO.agda	580
A.62	parallelSimple.agda	581
A.63	prefix.agda	587
A.64	preFix.agda	589
A.65	primitiveProcess.agda	590
A.66	process.agda	592
A.67	process2OptimizedProcess.agda	594
A.68	process2OptimizedProcess2.agda	595
A.69	process2OptimizedProcess3.agda	596
A.70	proofAss.agda	597
A.71	proofAssExt.agda	615
A.72	proofBisimForInterleaving.agda	652
A.73	proofBisimSFFdiMonadicLaws.agda	660
A.74	proofBisimSFFdiMonadicLaws2.agda	675
A.75	proofCommutativeExternalChoice.agda	678
A.76	proofEqforParSym.agda	686
A.77	proofEqforParSymTheoOnly.agda	687
A.78	ProofLawsSeq.agda	688
A.79	ProofLawsSeqTheoOnly.agda	696
A.80	proofMonadicLaw.agda	704
A.81	proofMonadicLawTheoremsOnly.agda	713
A.82	proofRefLaw.agda	724
A.83	proofRefLawFdiCorrected.agda	725
A.84	proofRefLawFdiModified.agda	728
A.85	proofRenamingSkip.agda	730
A.86	proofSymExt.agda	731
A.87	proofSymInterleaving.agda	735
A.88	proofSymInterleavingTheoOnly.agda	738
A.89	proofSymParPartone.agda	742

A.90	proofSymParR.agda	747
A.91	proofTerHide.agda	752
A.92	proofTerInter.agda	754
A.93	proofTerPar.agda	756
A.94	rec.agda	759
A.95	RefWithoutSize.agda	760
A.96	renamingOperator.agda	762
A.97	renamingProcess.agda	764
A.98	renamingResult.agda	765
A.99	restrict.agda	769
A.100	sequentialComposition.agda	771
A.101	sequentialCompositionRev.agda	772
A.102	showFunction.agda	774
A.103	showFunctionForSimulator.agda	778
A.104	showLabelP.agda	781
A.105	simulator.agda	782
A.106	simulatorCutDown.agda	786
A.107	skip.agda	788
A.108	test.agda	789
A.109	theoremProverAgdaChapterCod.agda	789
A.110	theoremProverAgdaChapterCod2.agda	803
A.111	theoremProverAgdaChapterCod3.agda	817
A.112	traceEquivalence.agda	819
A.113	traceImpliesTraceP.agda	820
A.114	TraceWithNextProcess.agda	826
A.115	TraceWithoutSize.agda	831
A.116	trainExample.agda	833
A.117	trainExampleCorrected.agda	850
A.118	trainExampleCorrectedOptimized.agda	853
A.119	trainExampleCorrectedOptimized2.agda	855
A.120	trainExampleCorrectedOptimized3.agda	857
A.121	trainExampleCorrectedOptimized3ProofNonRef1.agda	858
A.122	trainExampleCorrectedOptimized3ProofNonRefCutDown.agda	925
A.123	trainExampleOptimized3.agda	927
A.124	trainExampleOptimized3ProofRefinement.agda	928
A.125	trainExampleOptimized3ProofRefCutDownForPaper.agda	931
A.126	UnitModule.agda	932



Abstract

We introduce a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. We will enhance processes by a monad structure. The monad structure facilitates combining processes in a modular way, and allows to define recursion as a direct operation on processes. Processes are defined coinductively as non-well-founded trees. The nodes of the tree are formed by a atomic one step relation, which determines for a process the external, internal choices, and termination events it can choose, and whether the process has terminated. The data type of processes is inspired by Setzer and Hancock’s notion of interactive programs in dependent type theory. The operators of CSP will be defined rather than atomic operations, and compute new elements of the data type of processes from existing ones.

The approach will make use of advanced type theoretic features: the use of inductive-recursively defined universes; the definition of coinductive types by their observations, which has similarities to the notion of an object in object-oriented programming; the use of sized types for coinductive types, which allow coinductive definitions in a modular way; the handling of finitary information (names of processes) in a coinductive settings; the use of named types for automatic inference of arguments similar to its use in template Meta-programming in C++; and the use of interactive programs in dependent type theory.

We introduce a simulator as an interactive program in Agda. The simulator allows to observe the evolving of processes following external or internal choices. Our aim is to use this in order to simulate railway interlocking system and write programs in Agda which directly use CSP processes.

Then we extend the trace semantics of CSP to the monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. Because of the monadic settings, some adjustments need to be made to these laws.

Next we implement the more advanced semantics of CSP, the stable failures semantics and the failures divergences infinite traces semantics (FDI), in CSP-Agda, and define the corresponding refinement and equality relations. Direct proofs in these semantics are cumbersome, and we develop a technique of showing algebraic laws in those semantics in an indirect way, which is much easier. We introduce divergence-respecting weak bisimilarity and



strong bisimilarity in CSP-Agda, and show that both imply equivalence with respect to stable failures and FDI semantics. Now we show certain algebraic laws with respect to one of these two bisimilarity relations. As a case study, we model and verify a possible scenario for railways in CSP-Agda and in standard CSP tools.





Acknowledgement

First and foremost, I would like to thank my supervisor Anton Setzer. It has been such a pleasure and an honour to be his PhD student. I have learned so much from him, both academically and competently. He has guided me well and taught me how to carry out a PhD project. I appreciate all his contributions and support in his time to help me with ideas, and his desire to make my PhD experience productive and motivating. The joy and enthusiasm he has for his research students was contagious and motivational for me, even during tough times in the pursuit of my PhD. I am also thankful for the excellent example he has provided as a successful supervisor.

I would like to thank the internal examiner, John Tucker, and the external examiner Conor McBride, for creating a very detailed report on my thesis which contained lots of valuable suggestions for improvement.

Special thanks to his Majesty Abdullah II Ibn Al Hussein King of the Hashemite Kingdom of Jordan for his continued support and sponsorship which has inspired me to achieve excellent results in my PhD, as his Father King Hussain has supported my father during his PhD. I hope that I will make my country and family proud. It's such a privilege which I have always cherished and which has inspired me for continues improvement not only in my studies but also in my career.

The members of the department of computer science at Swansea University have contributed immensely to my personal and professional time at Swansea University. The group has been a source of companionship as well as good advice and collaboration. Specially I would like to thank Emma, Phil and Liam for their inspiring friendship and enthusiasm.

I am especially grateful and mainly indebted for the support of my parents who supported by me from primary school to the point of achieving a PhD. My parents raised me with love and passion for computer science and supported me in all my pursuits. I want to thank them for all their love and encouragement.





I would like to acknowledge my siblings who supported me throughout my life. I very much appreciate their keenness and inspiration. Especially, I would like to thank my sister Diana, Farah, and my brothers Shaker and Hashim for all for their love, support and encouragement during the final stages of my PhD.

Finally, I would like to thank my friends who have always supported me and for their ever-ending loyal friendship. All my friends, and especially Rashid AlJaraidah, Dr Foaz Irikat , Mohammad Tahir and Dr Ali Abdullah Allwan have helped me tremendously.

Thank you to everybody.



Chapter 1

Introduction

1.1 Motivation

The starting point of this work was the modelling of processes of the European Railway Train Management System (ERTMS) in the process algebra CSP. Having expertise in modelling railway interlocking systems in Agda (PhD project by Kanso [2012], Kanso and Setzer [2014]), we thought that an interesting step forward would be to model CSP in the interactive theorem prover and dependently typed programming language Agda. Our aim was to simulate railway interlocking system and write programs in Agda, which directly use CSP processes. A first step towards this project was the development of the library CSP-Agda Igried and Setzer [2016a,b]. CSP-Agda represents CSP processes coinductively and in monadic form. In CSP-Agda a monadic extension of CSP was developed, which is based on the IO monad. The IO monad allows to development of programs in a modular way using the bind construct. Interactive programs return a value when they terminate. The bind construct allows to sequentially compose a program with return value with a program, which depends on that return value.

The purpose of this thesis is to introduce CSP semantics in Agda, and carry out proofs of algebraic laws and of properties of example processes in CSP-Agda.

1.2 CSP

Communicating Sequential Processes (CSP) Hoare [1978], Roscoe [1998] is a formal specification language which was developed in order to model concurrent systems through their communications. It was developed by Hoare in 1978 Hoare [1978]. It is a member of the family of process algebras.

Process algebras are one of the most important concepts for describing

concurrent behaviours of programs. CSP has been used for modelling industrial systems and is supported by several industrial strength tools. Therefore, we thought that it would be of great benefit to integrate CSP into the theorem prover and dependently typed programming language Agda, in order to develop a methodology for programming concurrent systems in dependent type theory. This would allow as well to prove properties such as safety and liveness of CSP-processes in Agda, and to integrate tools for CSP into Agda.

1.3 The IO Monad in Agda

The model of CSP in Agda we are developing is essentially a variant of the IO monad which was formulated in Agda. In functional programming a lot of work has been invested in developing concepts for defining interactive, usually sequential programs. The main approach is Moggi's IO monad. The IO monad was developed by Moggi [1991]. It was pioneered by Wadler and Peyton Jones (Wadler [1990, 1995], Peyton Jones and Wadler [1993], Wadler [1998, 1997]) as a paradigm for representing IO in functional programming, especially Haskell. Hancock and Setzer [2000a, 1999, 2000b] have developed a version of the IO monad in dependent type theory, which for the sake of brevity we call in this thesis HS-monad. The HS-monad has been used together with other ideas for formalising IO in Idris (Brady [2008, 2013]). The HS-monad is now integrated into the standard library of Agda (Agda Community [2017a]) and into Idris. The HS-monad covers currently only sequential programs. In this thesis we explore the representation of processes in dependent type theory as a step towards concurrent interactive programs in dependent type theory.

The idea of the IO monad is that an element of $(\text{IO } A)$ is an interactive program, which may or may not terminate, and if it terminates returns an element of type A . We can use the monadic bind to compose a $p : \text{IO } A$ with a function $f : A \rightarrow \text{IO } B$ to form an element of $(\text{IO } B)$. The program is executed by first running p . If p terminates with result a , one continues running $(f \ a)$. Using nicely defined syntactic constructs, one can write sequences of operations in a way which looks similar to sequences of assignments in imperative style programming languages.

The IO monad has been used to develop interfaces and objects in object-based programming: Objects are server side interactive programs, which receive as commands method calls and return the result of this method call. Anton Setzer [2006] has used this approach in order to develop the notion of objects in dependent type theory. Together with Abel and Adelsberger (Abel et al. [2017]), he has extended this substantially to the library `ooAgda`

(Abel et al. [2016]) for objects in Agda. This library includes state dependent objects, server-side programs, and correctness proofs. It was used in order to develop graphical user interfaces in Agda.

We will present as well an executable interactive program in Agda, which simulates processes. Our vision is to use this approach for writing concurrent programs in Agda, similarly to as it is done in the Java library JCSP (Welch et al. [2007]). The main example we are investigating are processes in the context of the European Rail Traffic Management System (ERTMS [2013]), for which, as mentioned before, we have carried out some initial modelling in CSP. Our vision is that prototypes can be executed in Agda directly. Other examples one can envisage is to develop programs for networking in Agda.

1.4 Monadic Processes

In this thesis we introduce a new concept to process, namely that of a monadic processes which when terminating returns a value. This means that processes, when they terminate, return as well a return value. This facilitates the combination of processes in a modular way. We can take one process with return values of a given type, which call A , and another process which depends on A and has a return value of another type, say B , i.e. a function from A to processes with return value B . The monadic composition would first operate like the first process. If it returns with value $a : A$, the process operates like the second process instantiated with a .

An example would be a vending machine. We could define a first process corresponding to the insertion of money until a key is pressed. The return value would be the amount of money inserted, and the key pressed. Depending on this data, a second process can be defined, which finalises (or cancels) the vending process depending on the return value of the first process. The full vending machine is the result of combining those two processes using monadic bind.

The HS-monad reduces the IO monad to coinductively defined types. An element of $(\text{IO } A)$ is either a terminated program, or it is node of a non-well-founded tree having as label a command to be executed, and as branching degree the set of responses the real world gives in response to this command. In CSP-Agda we will model processes in a similar way. A CSP-Agda process can either terminate, returning a result. Or it can be a tree branching over external and internal choices,¹ where for each such choice a continuing process is given. So instead of forming processes by using high level operators, as it

¹There will be as well termination events, which we will discuss when introducing CSP-Agda.

is usually done in process algebras, our processes are given by these atomic one step operations. The high level operators are defined operations on these processes.

Since processes are defined coinductively, we can introduce processes directly corecursively without having to use the recursion combinator. Abel, Pientka, Thibodeau and Setzer have (in Abel et al. [2013], Setzer et al. [2014]) developed the notion of coinductive types as being defined by their elimination rules or observations. This notion has now been implemented in Agda. This has strong similarity to the notion of classes and objects in object oriented programming. Classes are essentially defined by their methods, and therefore given by their observations. Setzer, Abel and Adelsberger have used this approach in order to develop the notion of objects in dependent type theory (Setzer [2006], Abel et al. [2017]). In CSP-Agda we will make extensive use of this approach. Using a record type, we access directly for non-terminating processes the choice sets and corresponding subprocesses. It turns out that this makes programming with processes much easier, since it avoids the use of auxiliary functions.

We will make extensive use of sized types as introduced by Abel [2016] in Agda in the context of coinductive types. The main reason is that in its puristic form, primitive corecursion or guarded recursion doesn't allow to apply any functions to the corecursion hypothesis. With sized types size preserving functions can be applied and therefore coinductive programs be written in a modular way.

The index sets of processes will be given by universes, which are defined inductive-recursively as in Dybjer and Setzer [2003].

1.5 Proofs of Correctness of CSP-processes in Agda

Having developed processes in Agda, the next step is to prove properties about them. This requires developing CSP-semantics in CSP-Agda. In this thesis, we will introduce three main semantics of CSP into CSP-Agda: the traces, the stable failures, and the failures, divergences and infinite traces (FDI) model. Since we need to take care of return values and since we have a new notion of terminated processes, special considerations are needed. In trace semantics, return values need to be added to terminating traces. The algebraic laws of CSP need to be adapted as well to deal with the difference in return values. We will give examples of proofs in CSP-Agda. We note here that when proving equalities we refer to the equalities in the various

semantic models of CSP such as trace semantics, stable failures semantics etc, not with respect to the intensional equality type. We will define those semantic equalities as types in CSP-Agda.

1.6 Main Achievements

We aim to implement the process algebra CSP in dependent type theory specifically in theorem prover Agda. Specifically, we have achieved the following:

- We introduced a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. We enhanced processes by a monad structure. The monad structure facilitates combining processes in a modular way. Processes are defined coinductively as non-well-founded trees. This allows to define recursion as a direct operation of processes. The nodes of the tree are formed by an atomic one step relation, which determines for a process the external, internal choices, and termination events it can choose, and whether the process has terminated. The operators of CSP are defined rather than atomic operations, and compute new elements of the data type of processes from existing ones. We defined primitive processes such as STOP, SKIP and Div. Other operators such as prefix, external choice, internal choice, hiding, renaming, parallel, interleaving, interrupt and sequential composition are defined in Agda as well.
 - We have written a simulator in Agda. The simulator displays the process as a string. Then it computes and displays the set of \checkmark -events and their results, and of external and internal choices together with their labels, and allows the user to follow external and internal choices to continue with the next process obtained. This simulator is written in the same language Agda in which proofs about processes are carried out, avoiding any translations between different languages.
 - We have extended the trace semantics of CSP to the monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we have proven in CSP-Agda selected algebraic laws of CSP based on the trace semantics. We have as well shown correctness properties for a case study in the railway domain.
-

-
- We have implemented the stable failures and failures, divergences and infinite traces (FDI) semantics in Agda and defined the corresponding refinement and equality relations. Because of the monadic setting, some adjustments needed to be made to the algebraic laws. As an example, we prove that refinement with respect to stable failures semantics is a partial order.
 - We have extended the library CSP-Agda by implementing strong bisimilarity in order to facilitate proofs of algebraic properties for the trace and stable failure semantics.
 - We have as well implemented divergent-respecting weak (DRW) bisimilarity in Agda. We have shown that strong bisimilarity implies DRW bisimilarity, and both imply equivalence with respect to trace semantics, stable failures semantics and FDI semantics. This allows to carry out more easily proofs of algebraic properties in these semantics by showing first strong or DRW bisimilarity, and then obtaining directly equivalence in the other semantics. As an example, we apply this methodology to prove algebraic laws according to this semantics: We prove commutativity of the interleaving and external choice operators and prove two monadic laws.
 - We have carried out a case study of modelling an example from the railway domain in CSP. We used the FDR tools to check this model is free from deadlock, livelock, and to prove certain refinement statements. We used ProBE and CSP-Agda to simulate the possible scenario railway model, and proved as well refinement statements directly in CSP-Agda.

1.7 Thesis Outline

The remainder of this thesis is outlined as follows:

Chapter 2 introduces and reviews the process algebra CSP and presents its syntax, various semantics. We will as well discuss tool support for CSP.

Chapter 3 gives an overview of Agda. We will briefly present the main features of the theorem prover Agda, and discuss as well the reasons for choosing Agda for this project.

Chapter 4 gives an overview of related work presented in seven parts. First (Sect. 4.1), we give an overview of using formal methods in an industrial environment. Then (Sect. 4.2) we present related work regarding using process algebra in order to model and verify industrial strength systems. In Sect. 4.2.1, we discuss work related to CSP. In Sect. 4.3, we give an overview of theorem provers. In Sect. 4.4, we start to investigate work on using functional programming for defining processes algebras. In Sect. 4.5, we present work on defining process algebra in dependent type theory. In Sect. 4.6 work on defining process algebras in a coalgebraic manner is presented. In the following section 4.7, we investigate work on using the theorem prover Agda as a platform for modelling and verifying systems.

In Chapter 5 we introduce the library CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. We will introduce the operators of CSP in Agda in that chapter.

In Chapter 6 we introduce a simulator for CSP processes in Agda (called CSP-Agda-Simulator). This will be an interactive program in Agda. The simulator allows to explore the evolving of processes following external or internal choices.

In Chapter 7 we introduce trace semantics for CSP in Agda. This requires modifications to adjust trace semantics to the monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. To demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. Because of the monadic settings, some adjustments have made to these laws. All proofs and definitions have been type checked in Agda.

In Chapter 8 we implement stable failures semantics in CSP-Agda and define the corresponding refinement and equality relations. As before, because of the monadic setting, some adjustments need to be made.

In Chapter 9 we introduce as well failures, divergences and infinite traces semantics in CSP-Agda. As an example, we prove refinement with respect to stable failures semantics is a partial order.

In Chapter 10 we introduce strong and weak bisimulation and show that it implies trace and stable failure equivalence. As an example, we apply this methodology to prove algebraic laws for this semantics.

In Chapter 11 we present an example showing the use of the CSP-Agda. We model a small example of a railway interlocking system with three components, namely, Train, Signals and Segment. In the first stage, we model this scenario in CSP using ProBE tools, and verify this model against deadlock, livelock, and a refinement statement using the FDR 2 tool. It turns out that this model (which was deliberately chosen to demonstrate how mistakes can be detected) is not safe. Therefore we correct it and show that it is now correct using Probe and CSP-Agda. Finally, in Chapter 12 and 13 we summarise our work, and summarise future work related to this thesis.

1.8 Published Material

The work shown in this thesis is based on a sequence of publications at several conferences and workshops, coauthored by the originator of this thesis. These publications include:

1.8.1 Refereed Publications

Programming With Monadic CSP-style Processes in Dependent Type Theory (Igried and Setzer [2016a]). This article presents a first attempt to give the type theoretic interactive theorem prover Agda the ability to model concurrent programs by representing the process algebra CSP in monadic form. The set of processes forms a monad (Process A), which depends on a set A. This allows to define a dependent composition (monadic bind) and a dependent loop construct rec for processes. In this paper, we define processes coinductively. The termination checker of Agda guarantees productivity of processes. This allows defining processes recursively without having to reduce them to the recursion combinator. The processes in this paper are formed from an atomic one step iteration. The operators of CSP are defined operations, which combine processes defined from atomic operations. The paper also introduces a simulator as an interactive program in Agda. The simulator allows observing the evolving of processes following external or internal choices. The goal was to write programs in Agda which directly use CSP processes. The intended application domain is the simulation of railway interlocking system. Chapter 5 is mainly based on the results of this paper.

Trace and Stable Failures Semantics for CSP-Agda (Setzer and Igried [2017]). This paper reports on the continued development of a formalisation of the process algebra CSP in Agda. Compared to the work reported in the previous paper, this paper adds a formalisation of trace and stable failures semantics of the process algebra CSP in CSP-Agda. In CSP-Agda, CSP processes are in monadic form, which supports a modular development of processes. We introduce a variant of the definition of a trace, which records the process we obtain after following that trace. In this paper, we define as well the corresponding refinement and equality relations. As an instance, we prove commutativity of the external choice operator with respect to the trace semantics in CSP-Agda, and that refinement with respect to stable failures semantics is a partial order. All proofs and definitions have been type checked in Agda. Chapter 8 is mainly based on the results of this paper.

Defining Trace Semantics for CSP-Agda (Igried and Setzer [2018]). This paper is based on the library CSP-Agda, to which we have the trace semantics of CSP, an adjusted it to a monadic setting. Since in CSP-Agda processes are monadic, we need to record, in case a process has terminated after following a trace, the return value of this process. We implement this semantic, together with the corresponding refinement and equality relation, formally in CSP-Agda. We demonstrate the proof capabilities of CSP-Agda, by proving in CSP-Agda selected algebraic laws of CSP based on the trace semantics.

The examples covered in this paper are the laws of refinement, commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. All proofs and definitions have been type checked in Agda. Additional proofs of algebraic laws will be available in this thesis. Chapter 7 is mainly based on the results of this paper.

1.8.2 Refereed Short Papers

Modelling and Verification of RBC Handover Using CSP (Igried [2014]). In this paper, we are using the process algebra of communicating sequential processes (CSP) for modelling and verifying the RBC/RBC Handover. We used the FDR2 model checker to verify that it is free from Deadlock and Livelock.

Representing the Process Algebra CSP in Type Theory (Igried and Setzer [2016d]). In this paper introduce the library CSP-Agda. This

was the first step towards defining the process algebra CSP in Agda. CSP processes are implemented coinductively (or coalgebraically). Later in Igried and Setzer [2016a] we enhanced processes by defining them in monadic way.

Defining Trace Semantics for CSP-Agda (Igried and Setzer [2016c]).

In this paper, we gave the first definition of trace semantic for CSP-Agda. Then we showed how to prove algebraic laws of CSP in Agda using this semantics.

Strong Bisimilarity Implies Trace Semantics in CSP-Agda (Igried and Setzer [2016e]).

In this paper, we extend the library CSP-Agda with two semantics in order to prove properties of safety critical systems. We present trace semantics in the theorem prover Agda, together with the corresponding refinement and equality relation. To facilitate proofs of algebraic properties for this semantics, we introduce strong bisimilarity and show that it implies trace equivalence. As an example, we apply this methodology to commutativity of interleaving.

1.8.3 Papers in Preparation

Defining Strong Bisimulation for CSP-Agda. In this upcoming paper, we will define strong bisimulation and prove that strong bisimilarity implies trace semantics in CSP-Agda. We apply this methodology to the proof of algebraic laws according to the stable failures and trace model. Chapter 10 is mainly based on the results of this paper.

Weak Bisimilarity Implies Trace Semantics in CSP-Agda. In this paper, we will define divergent respecting weak bisimulation and prove that this form of weak bisimulation implies trace semantics in CSP-Agda. We apply this methodology to prove algebraic laws according to the trace model. Chapter 10 is mainly based on the results of this paper.

Weak Bisimilarity Implies Stable Failures Semantics in CSP-Agda.

In this paper, we will prove that weak bisimulation implies stable failures semantics. This methodology facilitates the proof of algebraic laws with respect to stable failures semantics. Chapter 10 is mainly based on the results of this paper.

Modelling and Verification of the ETCS Protocol In CSP-Agda.

In this planned paper we will investigate processes in the context of the

European Rail Traffic Management System ERTMS [2013], for which initial modelling in CSP was carried out in Igried [2014]. Our plan is that prototypes can be executed in Agda directly. Other examples one can envisage is to develop programs for networking in Agda. Chapter 11 presents a smaller case study, gives first result of this project.

1.8.4 Conferences, Workshop & talk

Attendance of workshop with refereed contributed talk at 26th Nordic Workshop on Programming Theory, NWPT '14, 19 - 31 Oct. 2014, Halmstad University, Sweden. Title of talk: Modelling and Verification of RBC Handover Using CSP

Attendance of workshop “BCS FACS - ProCoS Workshop on Provably Correct Systems”, 9 - 10 March 2015, London

Attendance of 22nd International Conference on Types for Proofs and Programs, TYPES 2016, 23-26 May 2016. Novi Sad, Serbia. Title of talk: Representing the Process Algebra CSP in Type Theory.

Attendance of Workshop on Type-Driven Development (TyDe 2016), Sun 18 - Sat 24 September 2016 Nara, Japan. Title : Programming with monadic CSP-style processes in dependent type theory.

Attendance of Workshop on Mathematical Logic and its Applications, Kyoto, Japan, 16 - 22 September 2016.

Attendance of Workshop on Coalgebra, Horn Clause Logic Programming and Types Edinburgh, UK, 28-29 November 2016 CoALP-Ty'16. Title: Defining Trace Semantics for CSP-Agda.

Attendance of 23rd International Conference on Types for Proofs and Programs, TYPES 2017. Title: Strong Bisimilarity Implies Trace Semantics in CSP-Agda.

Attendance of Proof, Verification and Complexity Seminar, Dept of Computer Science, Swansea University. Title of talk “Modelling and Verification of RBC Handover Using CSP”, 16th Oct. 2014.

Attendance of SPES_XT (Software Platform Embedded Systems) summer school , 17 - 23 Sept. 2014, Twente, The Netherlands. Title of talk: Formal Verification of CSP.

Talk given at JAIST, Japan ,September 2016 Kanazawa, Japan. Title: Programming with monadic CSP-style processes in dependent type theory.

Talk given at Computer Science Research Away Day 9th June in the Village hotel, Swansea, UK, 9th June 2017. Title: Defining CSP-style processes in dependent type theory.

Use of literal Agda. All displayed proofs in this thesis have been written using literal Agda (Agda Community [2017b]), which allows to combine Agda with \LaTeX code. They have been type checked in Agda. Full versions can be found in the repository of CSP-Agda (Igried and Setzer [2016b]).

Chapter 2

CSP

“Process algebras” were initiated in 1982 by Bergstra and Klop [1982] in order to provide a formal semantics to concurrent systems. A “process” is a representation of the behaviour of a concurrent system. “Algebra” means that the system is dealt with in an algebraic and axiomatic way (Baeten et al. [2007]). Process algebras allow to study distributed or parallel systems in an algebraic way. Processes are formed from a collection of operator symbols, and axioms express the properties of the processes formed from operators in terms of the properties of the processes used. Most process algebras have basic operators to construct finite processes, synchronisation and parallel constructs to express concurrency, and a notion of recursion to obtain infinite behaviour. The basis of process algebra is that it forces an equational logic on process terms, such that we can say two processes are equal if and only if the behaviour graphs for both processes are equal. The main process algebras are Calculus of Communicating Systems (*CCS*) (Milner [1982]), Communicating Sequential Processes (*CSP* Brookes et al. [1984]) and Algebra of Communicating Processes (*ACP*, Bergstra and Klop [1984b]), from which more advanced process calculi such as Milner’s π -calculus (Milner et al. [1992]) were derived.

In this thesis we will work on CSP. The main reason for choosing CSP is that it has considerable tool support and seems to be the process algebra which is most widely used in industry. This was very beneficial when modelling elements of the European Train Management System ERTMS in CSP. We could build as well on rich expertise on CSP at the computer science department at Swansea University.

The general technique of modelling processes coinductively based on a one step operation in a monadic way should work for other process calculi as well, we mainly investigated CCS where this should be possible. Some aspects might even simplify, since CCS doesn’t have termination events which turned out to be quite difficult to deal with with many subtle issues. In CCS

we could directly work with weak bisimilarity which is its main semantics, without the need to study the other semantics as in this thesis.

2.1 Communicating Sequential Processes

The process algebra CSP (Hoare [1978], Roscoe [1998], Schneider [1999]) was developed by Hoare in Hoare [1978]. CSP is a formal specification language, developed in order to describe concurrent systems by identifying their behaviour through their communications. CSP is a notation for studying processes which interact with each other and their environment. In CSP, we can describe a process by the way it can communicate with its environment. The most fundamental object in CSP is an event. Events can be external events, which can be observed externally and are given by a label, silent internal events, which are not observable from the outside, and termination events, corresponding to the termination of a process. The set of labels forms an alphabet Σ . A system is formed by one or more processes, where these processes interact with each other through their external and termination events. The overall system in CSP can be described by specifying the behaviour of interacting processes. CSP has a variety of semantics which give meaning to the processes.

Similar to CSP, the Calculus of Communicating Systems (CCS) notation Milner [1982], introduced in 1980 by Robin Milner, deals with interacting behaviours of the finite state machine. In CCS processes are defined as agents and the behaviour of processes is defined in terms of events they can perform. In CCS a labelled transition system is used to interpret the expressions of the language. Semantic equivalence is defined as bisimilarity, whereas semantic equivalence in CSP is based on extensions of trace semantics. The process algebra CCS, unlike CSP, CCS does not distinguish between external and internal choice operator, whereas CSP has two kinds of operators for choices, namely the external choices operator and the internal choices operator. In CSP the internal event never appears in trace semantics, by contrast, internal events in CCS appear in the trace semantics. In CSP there is a process STOP which represents deadlock and a process SKIP which represents successful termination. On the other hand, CCS does not distinguish between deadlock and successful termination. Instead it has a single “terminated” process, nil, which stands for both successful termination and deadlock.

In the following we will first present the syntax of CSP and then introduce different semantics for it.



2.2 Framework of CSP

We will first introduce the language of CSP operationally, which is defined in terms of how CSP processes are to be executed. By nature the language of CSP is denotational in nature (Schneider [1999], p. viii). Schneider in Schneider [1999] elucidates the language of CSP as well operationally, in order to obtain a better understanding of the CSP operators. There he introduces the transitions, events, processes, together with inference rules for deriving those transitions.

2.2.1 Transitions

The transition systems represent the behaviour of a process. Processes in CSP form a labelled transition system, where the one step transition is written as

$$P \xrightarrow{e} Q \quad \text{where } P, Q \text{ are processes and } e \text{ is an event.}$$

which means that process P can evolve to process Q by event e . The event e can be a label, it can be the silent transition τ , or it can be a termination event \checkmark . For example the execution of the process $a \longrightarrow b \longrightarrow STOP$ can be described by the LTS:

$$(a \longrightarrow b \longrightarrow STOP) \xrightarrow{a} (b \longrightarrow STOP) \xrightarrow{b} STOP$$

2.2.2 Inference rules

The inference rules of CSP are determined by inference rules of the form

$$\frac{\text{Antecedent}_1 \quad \dots \quad \text{Antecedent}_n}{\text{Conclusion}} \text{ (Sidecondition)}$$

Derivations of a transition are the derivations formed from those inference rules and are inductively defined.

The operational semantics of CSP defines processes as states. Transitions between states are the firing rules for those processes.

2.2.3 Events

A system in CSP is given by its components (processes). These components can interact with each other or with the environment by their interfaces. For instance, a cassette recorder might be considered as a process. Its interface will include buttons like *Play*, *Stop*, *Forward*, *Backward* etc., and as well



the door through which the tape can be inserted. Through this interface the process can be interact with the users (environment). This interface is considered as a set of events. The set of events in a CSP process interface are classified as a static specification, and considered as dynamic specifications by describing how it behaves operationally.¹ The events in CSP represent interactions, and are considered to be indivisible and instantaneous. The events in CSP could be atomic. For instance, the interface of the last example, the cassette recorder, has atomic events *Play*, *Stop*, *Forward*, *Backward*, etc. Events in CSP can be as well compound, i.e. it can have some structure. For instance, consider the Automated Teller Machine (ATM) as a process, where the customer can enter the card and return it, by ignoring the other events. This can be written in CSP as having the events *card.in* and *card.out*. Additionally, event can have some structure by a communication channel which carries some information (i.e. messages). The value v communicated over a channel c , can be described as an event $c.v$ in the process interface. For instance, consider the processes P with input channel *in* which can carry the values 0 and 1. Both values 0 and 1 carried over the channel c may appear in the interface of the process P as separate events *in.0* and *in.1*. Furthermore an event can be considered as an input and output event. For instance, consider the following CSP expression:

$$c!v \longrightarrow P$$

The above process is willing to output the value v over channel c , and then behave like a process P . The behaviour is captured by the following firing rule:

$$\frac{}{(c!v \longrightarrow P) \xrightarrow{c.v} P}$$

Input phenomena can be explained through the following expression:

$$c?x : T \longrightarrow P(x)$$

Here a processes $P(x)$ is defined for each $x \in T$. The process $c?x : T \longrightarrow P(x)$ is willing to received any value x of the type T over the channel c , and then behave like a process $P(x)$. The receiving phenomena is described in CSP by the following firing rule:

$$\frac{}{(c?x : T \longrightarrow P(x)) \xrightarrow{c.v} P(v)} [v \in T]$$

¹We note here that in CSP-Agda we will index the possible events by index sets. These index sets will not carry any semantic meaning.

Besides this kind of external events, which are labelled by elements of an alphabet Σ , there are internal events labelled by τ , which are internal changes of the state of a process. Furthermore there are termination events \checkmark , which will be introduced in Subsect. 2.3.1 below.

In this thesis, we assume an alphabet Σ , where $\checkmark, \tau \notin \Sigma$. μ will be an element of $\Sigma^{\checkmark, \tau} := \Sigma \cup \{\checkmark, \tau\}$, and a, b, c will be elements of Σ .

2.3 The Syntax of CSP

In this section we will explain each operator and give an example to illustrate its meaning. In the following table, we list the syntax of CSP process i.e. the primitive CSP processes and the CSP operations. Here Q represents CSP processes:

$Q ::=$	$STOP$	deadlock process
	$SKIP$	terminating process
	DIV	diverging process
	$a \longrightarrow Q$	event prefix process
	$Q \square Q$	external choice
	$Q \sqcap Q$	internal choice
	$Q \setminus a$	hiding
	$Q[R]$	renaming
	$Q \parallel [A \mid B] Q$	alphabetised parallel
	$Q \parallel\!\!\parallel Q$	interleaving
	$Q \parallel [A] Q$	interface parallel
	$Q \triangle Q$	interrupt
	$Q; Q$	composition

In the above table, A and B are parameters, which represent a set of events that Q_{left} and Q_{right} are respectively allowed to perform. Furthermore, R is a function for renaming event in the process Q .

2.3.1 Primitive Processes

CSP has many primitive, i.e. atomic processes. The main primitive process is $STOP$. The process $STOP$ does not have any transition rule since it can not engage in any event and refuses all communication. The process $STOP$ represents the deadlock process, which has no (internal or external) transitions.

CSP has as well the process $SKIP$, which represents successful termination (i.e. reached terminations). This is indicated by performing a special

event called the termination event \checkmark . As the \checkmark is a special event, it can not be a member of the alphabet of the process (i.e., it can not be appear as a part of prefix operator). The CSP process *SKIP* terminates immediately. It has the following rule:

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

Furthermore we have the diverging *DIV*. This process has as only transition

$$\frac{}{DIV \xrightarrow{\tau} DIV}$$

which means it can perform an infinite sequence of τ -transitions.

2.3.2 Sequential Composition

The sequential composition of two processes P and Q is written in CSP as $P ; Q$. The combination $P ; Q$ behaves as P until it terminates, in which case it switches to Q . For instance, the process

$$COMP_1 = a \longrightarrow SKIP ; b \longrightarrow STOP$$

perform the event a followed by a b event and then behaves like a process *STOP*, i.e. it deadlocks. This process behaves as $a \rightarrow b \rightarrow STOP$. The following processes

$$COMP_2 = a \longrightarrow STOP ; b \longrightarrow STOP$$

behaves in different way: after performing event a the process behaves like a process *STOP* (i.e. deadlock), which never terminates successfully. Therefore control is never passed over to process $b \longrightarrow STOP$, so $COMP_2$ behaves as $a \longrightarrow STOP$.

The transition rules for the sequential composition operator are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P}}{P ; Q \xrightarrow{\tau} Q} \quad \frac{P \xrightarrow{\mu} \bar{P}}{P ; Q \xrightarrow{\mu} \bar{P} ; Q} [\mu \neq \checkmark]$$

The sequential composition operates initially like the first process. When it terminated, the successful termination event becomes internal, and after that internal event the process behaves like Q . The process composed by sequential composition has only a termination event once it has reached a termination event of the 2nd process.

2.3.3 Event Prefix

The prefix operator constructs from an existing process P a process $a \longrightarrow P$, pronounced in CSP 'a then P'. It has one external event $a \in \Sigma$ after which it behaves like P . Its behaviour is captured by the following firing rule:

$$\frac{}{(a \longrightarrow P) \xrightarrow{a} P}$$

This is an example of an axiom, i.e. a rule with no side conditions or antecedents.

For instance, consider then process $PreF$ which can perform only an event a and then behave like the process $STOP$ having no transitions. It is given as follows:

$$PreF = a \longrightarrow STOP$$

2.3.4 Recursion

CSP uses recursive definitions in order to describe the execution of processes, which operate in an infinite manner. Recursive processes are introduced in CSP in an equational style $N = P$. Here N is a new name for the process to be defined, and P is a CSP expression which can make use of P . This recursive definition of processes will work only if the righthand side of the equation begins with at least one event prefixed to all recursive occurrences of the process name. A example where this condition is not fulfilled would be the recursive equation $N = N$, which doesn't define a valid process. A process which begins with a prefix is called a guarded process, and if the right hand side of a recursive definition is a guarded processes the recursive definition has a unique solution.

For example $P = a \longrightarrow b \longrightarrow P$ is a process that repeats infinitely often an a event followed by an b . The firing rule for unwinding a process N recursively bound to a definition of process P is given as follows:

$$\frac{P \xrightarrow{\mu} \bar{P}}{N \xrightarrow{\mu} \bar{P}} [N = P]$$

The rule above states that the transitions from P are the same as those originating from N . We can write the process P in the form of $F(N)$ in order to make the dependency on N explicit. Then we have $F(Y) = P[Y/N]$ where the latter is the substitution of N by Y . As an example we have

$$a \longrightarrow b \longrightarrow P [a \longrightarrow STOP/P] = a \longrightarrow b \longrightarrow a \longrightarrow STOP$$

2.3.5 Internal Choice

CSP offers two kinds of operators for choices between two processes: external choice and internal choice. In external choice the operator allows the environment to choose an external event of P or Q , and then continue as process P or Q , respectively. In contrast, internal choice leaves the choice between processes to the process. The internal choice between processes P and Q is written as $P \sqcap Q$. For this processes the process chooses non-deterministically to continue as P or as Q . Consider for instance the process

$$IntCH = a \longrightarrow SKIP \sqcap b \longrightarrow STOP$$

In this case the process switches internally to either $a \longrightarrow SKIP$ or $b \longrightarrow STOP$. In the first case it has only one external event, namely an a -event, and then behaves like $SKIP$ (i.e. successfully termination), or it has only one b event then behaves like $STOP$ (i.e. deadlock). The inference rule for the internal choice operator as follows:

$$\frac{}{P \sqcap Q \xrightarrow{\tau} \bar{P}} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} \bar{Q}}$$

As for external choice, CSP has an indexed internal operator, which take an arbitrary number of processes. For instance the process

$$\sqcap_{x \in \{a \ b \ c\}} x \longrightarrow STOP$$

can internally choose $x = a$, or $x = b$ or $x = c$, and then have an external event labelled by x followed by the $STOP$ process.

The inference rule for the indexed internal choice operator is as follows:

$$\frac{}{\sqcap_{i \in J} P_i \xrightarrow{\tau} P_j} [(j \in J)]$$

2.3.6 External Choice

External choice between processes P and Q is written in CSP as $P \sqcup Q$. External choice allows the environment to choose an external event of P or Q , and then continue as process P or Q , respectively. For instance, consider the process $ExtCH$

$$ExtCH = a \longrightarrow SKIP \sqcup b \longrightarrow STOP$$

The $ExtCH$ process can firstly perform one of two event a or b . If a is chosen it continues as $SKIP$, and if b is chosen it continues as $STOP$. The firing rule for the external choice are as follows:

$$\begin{array}{c}
\frac{P \xrightarrow{a} \bar{P}}{P \sqcap Q \xrightarrow{a} \bar{P}} \\
Q \sqcap P \xrightarrow{a} \bar{P}
\end{array}
\qquad
\begin{array}{c}
\frac{P \xrightarrow{\tau} \bar{P}}{P \sqcap Q \xrightarrow{\tau} \bar{P} \sqcap Q} \\
Q \sqcap P \xrightarrow{\tau} Q \sqcap \bar{P}
\end{array}$$

There exist as well a generalisation of the binary external choice operator, which creates the external choice over a finite number of processes indexed over an index set. The firing rules for the indexed external choice operator are as follows:

$$\frac{P_j \xrightarrow{a} \bar{P}}{\sqcap_{i \in I} P_i \xrightarrow{a} \bar{P}} [j \in I] \qquad
\frac{P_j \xrightarrow{\tau} \bar{P}_j}{\sqcap_{i \in I} P_i \xrightarrow{\tau} \sqcap_{i \in I} \bar{P}_i} [j \in I]$$

2.3.7 Parallel and Interleaving

CSP has several parallel operator which enforce a number of processes to work together and interact through synchronous events. When the participants engage in the synchronous events, synchronisation occurs simultaneously, which is like a handshake, and therefore called *handshake synchronisation*. In the following we will explain the most important parallel operators together with examples.

Alphabetised Parallel

The alphabetised parallel operator of CSP depends on alphabets A, B for the two processes P, Q operating in parallel. When put in parallel, process P can only perform events in A and process Q can only perform events in B . Furthermore, any event in $A \cap B$ needs to be synchronised between P and Q . Silent transitions can occur individually in P and in Q , whereas the termination event needs to be synchronised between P and Q . The transition rules are as follows:

$$\frac{P \xrightarrow{a} \bar{P} \quad Q \xrightarrow{a} \bar{Q}}{P \parallel [A \mid B] \parallel Q \xrightarrow{a} \bar{P} \parallel [A \mid B] \parallel \bar{Q}} [a \in (A \cup \{\checkmark\}) \cap (B \cup \{\checkmark\})]$$

$$\frac{P \xrightarrow{\mu} \bar{P}}{P \parallel [A \mid B] \parallel Q \xrightarrow{\mu} \bar{P} \parallel [A \mid B] \parallel Q} [\mu \in (A \cup \tau) \setminus B]$$

$$Q \parallel [B \mid A] \parallel P \xrightarrow{\mu} Q \parallel [B \mid A] \parallel \bar{P}$$

The first rule states that events in the interaction between A and B and termination events need to be executed by both processes simultaneously.

The second rule allows each process to perform events, which are not in the alphabet of the other process and silent transitions individually.

As an example consider

$$(a \longrightarrow P) \parallel [\{a\} \mid \{a, b\}] \parallel (a \longrightarrow Q \sqcap b \longrightarrow Z)$$

This process can perform the event a and behave as a process $P \parallel [\{a\} \mid \{a, b\}] \parallel Q$, or perform the event b and behave as $(a \longrightarrow P) \parallel [\{a\} \mid \{a, b\}] \parallel Z$.

Consider consider the following process:

$$(a \longrightarrow P) \parallel [\{a, b\} \mid \{a, b\}] \parallel (a \longrightarrow Q \sqcap b \longrightarrow Z)$$

It cannot perform event b , since the left process cannot perform it. It can only perform event a and then behave as a $P \parallel [\{a, b\} \mid \{a, b\}] \parallel Q$.

Interleaving

Independent concurrent behaviour is represented in CSP by the interleaving operator. The interleaving of P and Q is written in CSP as $P \parallel\parallel Q$ and is pronounced as “ P interleave Q ”. The combination $P \parallel\parallel Q$ executes processes P and Q independently. The only way the processes can engage with each other is when they terminate. The transition rules are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P} \quad Q \xrightarrow{\checkmark} \bar{Q}}{P \parallel\parallel Q \xrightarrow{\checkmark} \bar{P} \parallel\parallel \bar{Q}}$$

$$\frac{P \xrightarrow{\mu} \bar{P}}{P \parallel\parallel Q \xrightarrow{\mu} \bar{P} \parallel\parallel Q} \quad [\mu \neq \checkmark]$$

$$Q \parallel\parallel P \xrightarrow{\mu} Q \parallel\parallel \bar{P}$$

If P and Q have disjoint alphabets A, B , the alphabetised parallel behaves as interleaving: If P and Q have events in A, B respectively, and $A \cap B = \emptyset$ then $P \parallel [\{A\} \mid \{B\}] \parallel Q$ and $P \parallel\parallel Q$ behave in the same way.

For instance

$$(a \longrightarrow b \longrightarrow P) \parallel [\{a, b\} \mid \{c, d\}] \parallel (c \longrightarrow d \longrightarrow Q)$$

behaves as

$$(a \longrightarrow b \longrightarrow P) \parallel\parallel (c \longrightarrow d \longrightarrow Q)$$

Interface Parallel

The interface parallel operator defines the parallel operation of two processes, where a subset of the interface is synchronised. So the synchronised event can only occur when all processes are able to perform it. The operational semantics for the interface parallel operator is given by the following rules:

$$\frac{P \xrightarrow{a} \bar{P} \quad Q \xrightarrow{a} \bar{Q}}{P \parallel [A] Q \xrightarrow{a} \bar{P} \parallel [A] \bar{Q}} [a \in A \cup \checkmark]$$

$$\frac{P \xrightarrow{\mu} \bar{P}}{P \parallel [A] Q \xrightarrow{\mu} \bar{P} \parallel [A] Q} [\mu \notin A \cup \checkmark]$$

$$Q \parallel [A] P \xrightarrow{\mu} Q \parallel [A] \bar{P}$$

Consider for instance $P = a \longrightarrow b \longrightarrow P$, $Q = a \longrightarrow c \longrightarrow Q$. The only transition of $P \parallel [\{a\}] Q$ is

$$P \parallel [\{a\}] Q \xrightarrow{a} (b \longrightarrow P) \parallel [\{a\}] (c \longrightarrow Q)$$

After this transition it can continue in two different ways:

$$(b \longrightarrow P) \parallel [\{a\}] (c \longrightarrow Q) \xrightarrow{b} P \parallel [\{a\}] (c \longrightarrow Q) \xrightarrow{c} P \parallel [\{a\}] Q$$

and

$$(b \longrightarrow P) \parallel [\{a\}] (c \longrightarrow Q) \xrightarrow{c} (b \longrightarrow P) \parallel [\{a\}] Q \xrightarrow{b} P \parallel [\{a\}] Q$$

2.3.8 Hiding

The hiding operator, hides a set of events in a process, replacing them by internal τ -transitions. The main usage is when processes communicate with each other, and one wants to hide these internal communications. The notation in CSP for hiding is $P \setminus A$ pronounced “ P hides A ”, which hides the events in A from process P . Since the events are no longer visible, no other process can engage with them. The behaviour of the hiding operator is determined by the following firing rules:

$$\frac{P \xrightarrow{a} \bar{P}}{P \setminus A \xrightarrow{\tau} \bar{P} \setminus A} [a \in A] \quad \frac{P \xrightarrow{\mu} \bar{P}}{P \setminus A \xrightarrow{\mu} \bar{P} \setminus A} [\mu \notin A]$$

For example, the protocol stop-and-wait (based on examples taken from Schneider [1999] Section 3.1) implements a one-place buffer as a process consists of two parts *Sender* and *Receiver*. When its combined through the parallel operator a one-place buffer is formed. A messages input to the *Sender* is passed over the channel *mid* to the *Receiver*, then the *Sender* waits for acknowledgement. The processes *Sender* and *Receiver* are specified as follows:

$$\begin{aligned} \textit{Sender} &= \textit{in}?x \longrightarrow \textit{mid}!x \longrightarrow \textit{ack} \longrightarrow \textit{Sender} \\ \textit{Receiver} &= \textit{mid}?y \longrightarrow \textit{out}!y \longrightarrow \textit{ack} \longrightarrow \textit{Receiver} \end{aligned}$$

The *Sender* reads a value of type T over channel *in* and binds it to the variable x . Afterwards this value is sent over channel *mid* to *Receiver*. When the message has been output, *Receiver* sends an acknowledgement. The combined behaviour is obtained by combining processes *Sender* and *Receiver* using the parallel operator, and then hiding the internal communications, as follows:

$$(\textit{Sender} \parallel \textit{Receiver}) \setminus (\textit{mid} \cup \textit{ack})$$

In the above definition *Sender* and *Receiver* are combined using the parallel operator and all communication along channels *mid* and via event *ack* are hidden. The resulting process can be considered as a black box regarding events *mid* and *ack*. Externally it can receive value over a channel *in* and output it on channel *out*.

2.3.9 Renaming

The renaming operator allows to rename the events in a process. This operator is useful to obtain a new process by reusing previous description of processes without the need for rewriting the processes in full. In the language of CSP a total bijective function ($f : \Sigma \cup \checkmark \longrightarrow \Sigma \cup \checkmark$) on events is used to described the changing of the events, with the additional condition that $f(\checkmark) = \checkmark$. The renaming function can map external events to external events, but not to internal events. Termination events are not changed by the renaming function. The operational semantics of renaming operator is captured by the following transition rules:

$$\frac{P \xrightarrow{a} \bar{P}}{f(P) \xrightarrow{f(a)} f(\bar{P})} \quad \frac{P \xrightarrow{\tau} \bar{P}}{f(P) \xrightarrow{\tau} f(\bar{P})}$$

2.3.10 Interrupt

The interrupt operator $P \triangle Q$ offers the mechanism of passing the control from one process to another. The difference between $P \triangle Q$ and sequential composition is the control in $P \triangle Q$ can pass from P to Q by interrupt at any time (i.e. by Q stealing control). The interrupt operator in the process $P \triangle Q$ allows the control to remain with P while the events are carried out by P . The moment an external event is performed by Q , the control passes to Q . The inference rules for the interrupt operator are as follows:

$$\begin{array}{c} \frac{P \xrightarrow{\mu} \bar{P}}{P \triangle Q \xrightarrow{\mu} \bar{P} \triangle Q} [\mu \neq \checkmark] \qquad \frac{P \xrightarrow{\checkmark} \bar{P}}{P \triangle Q \xrightarrow{\checkmark} \bar{P}} \\[10pt] \frac{Q \xrightarrow{\tau} \bar{Q}}{P \triangle Q \xrightarrow{\tau} P \triangle \bar{Q}} \qquad \frac{Q \xrightarrow{a} \bar{Q}}{P \triangle Q \xrightarrow{a} \bar{Q}} \end{array}$$

2.4 The Semantics of CSP

The main goal of a semantics for CSP is to determine whether two processes P and Q are equal, and whether process P refines process Q . These two conditions are in fact equivalent, both can be formulated in terms of the other, and therefore it is only necessary to define one of them in a semantics. CSP has different kinds of semantics: operational semantics, denotational semantics, and axiomatic semantics. Each semantics gives a meaning to the expressions.

Operational semantics is used to create a transition system for a CSP process. The axiomatic semantics allows to derive facts from the derivation rules. It offers a set of algebraic laws to transform processes into other equal processes, and can be used to prove that two different processes are equal. Denotational semantics determines a semantic domain and for each process and element of this domain as its semantics. CSP has according to Schneider [1999] three main denotational semantics: (1) trace semantics (*traces*), (2) stable failures semantic (*failures*), and (3) failures, divergences, and infinite traces semantics (*failures, divergences, infinites*). Apart of this there exist three main kind of bisimulation semantics: strong bisimulation, weak bisimulation, and divergence-respecting weak bisimulation.

2.4.1 Trace Semantics

In CSP traces of a process are the sequences of actions, i.e. the labels of external choices, a process can perform. We simply record the actions that

a process may perform. For example

$$\langle \text{light_on}, \text{light_off} \rangle$$

is a single trace, where we firstly observe the light is on followed by the light is off. Since the processes in CSP are non-deterministic, a process can follow different traces during its execution.

The trace semantics of a process is the set of its traces. For instance, consider the Bus process, in which a person may wish to board the bus, pay for this services, and then alight in next stop. This process is represent in CSP as follows (based on examples taken from Schneider [1999]):

$$\begin{aligned} \text{Bus} = \text{board}.A \longrightarrow (\text{pay}.90 \longrightarrow \text{alight}.B \longrightarrow \text{Stop} \\ \square \text{alight}.A \longrightarrow \text{Stop}) \end{aligned}$$

The set of traces for the Bus services process is

$$\begin{aligned} \text{traces}(\text{Bus}) = \{ \langle \rangle, \\ \langle \text{board}.A \rangle, \\ \langle \text{board}.A, \text{pay}.90 \rangle, \\ \langle \text{board}.A, \text{pay}.90, \text{alight}.B \rangle, \\ \langle \text{board}.A, \text{alight}.A \rangle \} \end{aligned}$$

In CSP, a process P refines a process Q , written $(P \sqsubseteq_T Q)$ if and only if any observable behaviour of Q is an observable behaviour of P , i.e. if $\text{traces}(Q) \subseteq \text{traces}(P)$:

Two processes P, Q are equal with respect to trace semantics, written $P \equiv_T Q$, if they refine each other, i.e. if $\text{traces}(P) = \text{traces}(Q)$:

2.4.2 Stable Failures Semantics

The trace semantics refers only to the observable traces. It does not distinguish between external and internal choice. In particular it does not tell what a process can refuse to do. In case of external choice, a process cannot refuse any of the external choices available for the subprocesses, whereas in case of internal choice, it can switch internally to one of the subprocesses and reach a state, where it can carry out only the external choices of that subprocess. A good example can found in chapter 8. The stable failures mode has been developed to take care of this problem and to distinguish between external and internal choice.² The stable failures model refers to a refusal set. A

²See Sect. 8.5 of Roscoe [1998] for the precise history of the stable failures model.

refusal set is a set of events a process fails to perform, no matter how long it is offered. Failures in CSP are defined as a pair (t, X) , where $t \in \text{trace}(P)$ and X is a refusal set for process P after performing trace t . This means that X is a set of labels such that after performing trace t , the processes can reach a state where it cannot carry out a transition with label in X . A failure is called a stable failure, if the resulting process with this failure set cannot carry out any internal transition.

Refinement between two process in the Stable Failures semantics holds whenever it holds between their sets of traces and stable failures (written as $\text{failures}(P)$):

$$P \sqsubseteq Q \text{ iff } \text{traces}(Q) \subseteq \text{traces}(P) \wedge \text{failures}(Q) \subseteq \text{failures}(P)$$

2.4.3 Failures, Divergences, and Infinite Traces Semantics

The stable failures model records the events that a process performs with a set of events a process fails to perform after a process stabilises. The stable failures model is not effective in analysing processes which can diverge, which means they have an infinite sequence of τ -transitions. The stable failures model ignores any divergent behaviour.

Consider for instance process $P = \text{STOP}$ and process $Q = \text{STOP} \sqcap \text{DIV}$. the only trace for P and Q is $\langle \rangle$, the only stable state is STOP which refuses all events. So P and Q are equivalent with respect to the two previous semantics. However Q can diverge by choosing a τ -transition to DIV , whereas P can't.

There are two kinds of divergent behaviour:

- Traces that lead to a divergent state, i.e. a state where the process can only perform an infinite sequence of τ -transitions.
- Infinite traces in which a process can have infinitely many events, but may as well fork off into a divergent behaviour.

In the *Failures/Divergences/InfiniteTraces* model (*FDI*) of CSP, these behaviours are introduced alongside failures information. In this approach we can identify a process P with the failures, divergences, and infinite traces that may be observed. Since this approach takes account of divergent, infinite behaviour and as well stable failures, it is more discriminating than the stable failures semantics. The first set, referred to as the stable failures set, consists of a pair (t, X) , where t is a trace and X is a refusal set for the trace t . The second component is *divergence*, which consists of all traces which

leads to a divergent process. Infinite traces is the set of all infinite sequences of events from Σ , a process can perform.

2.4.4 Semantics of Recursive Processes

Isobe and Roggenbach [2005] gave two approaches for the underlying domain of the semantic domain of the stable failures semantics, in order to obtain fixed points of recursive processes: one is to use *complete metric spaces* (*cms*), where Banach's theorem shows the existence of fixed points. The other one is to use *complete partial orders*, and Tarski's theorem is used here to show the existence of fixed points. Whereas Banach's theorem guarantees uniqueness of the fixed point, Tarski's theorem does not guarantee uniqueness. In case of the *traces* model and *divergences* the least fixed point is chosen, whereas in case of the set of *failures* the largest fixed point is chosen, since this set is a negative property about what a process cannot do. Using induction over least and coinduction over largest fixed points one can show that one process refines another process.

2.4.5 Bisimilarity for CSP

The main notion for determining process equivalence in CSP is via traces, failures, divergences, etc. Those semantics don't refer directly to the underlying transition system. Other process algebras like CCS instead define the basic meaning of processes by using a labelled transition system (LTS). Using this approach one can determine equivalence by determining, which LTSs behave in the same way. Several equivalences over LTSs have been suggested. The most fundamental one is the notion of strong bisimulation. In strong bisimulation, two process are considered to be equivalent, if they have the same set of external, silent, and termination events available immediately, by these events leading to processes that are themselves equivalent. We first fix some notation:

Definition 2.4.1 (a) $\Sigma^\checkmark := \Sigma \cup \{\checkmark\}$ where we assume $\checkmark \notin \Sigma$.

(b) $\Sigma^{*,\checkmark}$ is the set of finite sequences of events, possibly followed by a \checkmark .

In CSP we can define strong bisimulation according to Roscoe [2010, 1998] as follows:

Definition 2.4.2 (a) The relation R on the set of nodes S' of the LTS S is a strong bisimulation iff the following hold:

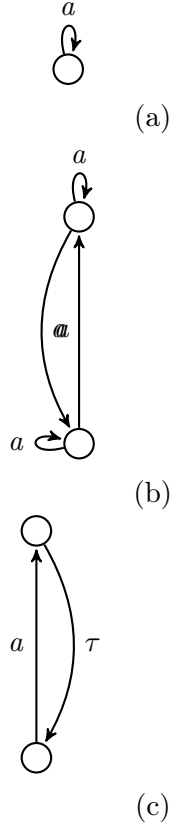


Figure 2.1: Unfolding LTSs
Roscoe [2010]

- $\forall P_1, P_2, Q_1 \in S'. \forall \mu \in \Sigma^{\vee, \tau}.$
 $P_1 R P_2 \wedge P_1 \xrightarrow{\mu} Q_1 \implies \exists Q_2 \in S'. P_2 \xrightarrow{\mu} Q_2 \wedge Q_1 R Q_2$
- $\forall P_1, P_2, Q_2 \in S'. \forall \mu \in \Sigma^{\vee, \tau}.$
 $P_1 R P_2 \wedge P_2 \xrightarrow{\mu} Q_2 \implies \exists Q_1 \in S'. P_1 \xrightarrow{\mu} Q_1 \wedge Q_1 R Q_2$

(b) *Bisimilarity is the largest bisimulation relation, or equivalently, the union of all bisimulation relations.*

In Fig 2.1 all the nodes of both systems 2.1a and 2.1b are strongly bisimilar. However, they are not strongly bisimilar to system 2.1c, since the system 2.1c can perform internal event (τ). This reveals the main weakness of strong bisimulation as a technique for analysing process behaviour, since bisimilarity distinguishes processes which differ in invisible events, although such processes are observationally equivalent. Other forms of bisimulation for LTSs have been proposed, most of them are based on weak bisimulation.

Weak bisimulation varies bisimulation by demanding that, if one process can make a possibly empty finite sequence of τ -transitions, or a possibly empty finite sequence of τ -transitions followed by an external event followed by another sequence of τ -transitions, or a sequence of τ transitions followed by a termination process, then the other process can do the same (where the number of τ -transitions might be different from the first process). It is still not suitable for CSP since it cannot distinguish between the primitive process *STOP* and *DIV*.

Divergence-respecting weak bisimulation is a slight strengthening of weak bisimulation in order to fix this problem by demanding in addition to the conditions of weak bisimilarity that, if one process is divergent, i.e. allows an infinite sequence of τ -transitions, the other one is divergent as well.

In order to define it we first define the following:

Definition 2.4.3 (a) $P \uparrow$, “ P is immediately divergent”, iff there exists a sequence $(P_i)_{i \in \mathbb{N}}$ such that $P = P_0$ and $\forall i \in \mathbb{N}. P_i \xrightarrow{\tau} P_{i+1}$.

(b) $P \xRightarrow{s} Q$ iff there exists P_0, \dots, P_m such that $P_0 = P, P_m = Q$ and events x_i such that $P_i \xrightarrow{x_i} P_{i+1}$, and s is the sequence of x_i such that $x_i \neq \tau$.

So $P \xRightarrow{s} Q$ means we can get from P to Q by following transitions in the trace s and in addition an arbitrary number of τ -transitions.

Following Roscoe [2010, 1998], we define in CSP *divergence-respecting weak bisimulation*, as follows:

Definition 2.4.4 (a) The relation R on the set of nodes S' of the LTS S is divergence-respecting weak bisimulation, in short *DRW-bisimulation*, iff $n R m$ implies the following

- $(P \uparrow \Leftrightarrow Q \uparrow)$
- $\forall P_1, P_2, Q_1 \in S'. \forall s \in \Sigma^{*, \checkmark}. P_1 R P_2 \wedge P_1 \xRightarrow{s} Q_1 \Rightarrow \exists Q_2 \in S'. P_2 \xRightarrow{s} Q_2 \wedge Q_1 R Q_2$
- $\forall P_1, P_2, Q_2 \in S'. \forall s \in \Sigma^{*, \checkmark}. P_1 R P_2 \wedge P_2 \xRightarrow{s} Q_2 \Rightarrow \exists Q_1 \in S'. P_1 \xRightarrow{s} Q_1 \wedge Q_1 R Q_2$

(b) Divergence-respecting weak bisimilarity, in short *DRW-bisimilarity*, is the largest *DRW bisimulation*, or equivalent the union of all *DRW bisimulations*.

(c) A relation R is a weak bisimulation iff it fulfils the same conditions as a *DRW-bisimulation*, except that $m \uparrow \Leftrightarrow n \uparrow$ is not required. *Weak bisimilarity* is the largest weak bisimulation relation.

Remark 2.4.5 *If we replace in Definition 2.4.4 the quantification over $\Sigma^{*,\vee}$ by $\Sigma^\vee \cup \{\langle \rangle\}$, we obtain an equivalent definition.*

We obtain (taken from Roscoe [2010, 1998]) the following key lemma for DRW-bisimulation:

Lemma 2.4.6 (Key-Lemma for DRW-bisimulation) *Let R be a DRW-bisimulation.*

- $\forall P, P', Q \in S'. \forall s \in \Sigma^{*,\vee}. P R P' \wedge P \xrightarrow{s} Q$
 $\Rightarrow \exists Q' \in S'. P' \xrightarrow{s} Q' \wedge Q R Q' \wedge (\text{stable}(Q) \Rightarrow \text{stable}(Q'))$
- $\forall P, P', Q' \in S'. \forall x \in \Sigma^{*,\vee}. P R P' \wedge P' \xrightarrow{s} Q'$
 $\Rightarrow \exists Q \in S'. P \xrightarrow{s} Q \wedge Q R Q' \wedge (\text{stable}(Q') \Rightarrow \text{stable}(Q))$

Proof We just prove the first direction. It suffices to show

$$P R Q \wedge \text{stable}(P) \Rightarrow \exists Q'. Q \xRightarrow{\langle \rangle} Q' \wedge P R Q' \wedge \text{stable}(Q') \quad (*)$$

Assume we have $(*)$, and assume

$$P_1 R P_2 \wedge P_1 \xrightarrow{s} Q_1 \wedge \text{stable}(Q_1)$$

Then by R being a DRW-bisimulation we obtain a Q'_2 such that

$$Q_1 R Q_2 \wedge P_2 \xrightarrow{s} Q_2$$

By $(*)$ we get that there exists Q'_2 such that

$$Q_1 R Q'_2 \wedge Q_2 \xRightarrow{\langle \rangle} Q'_2 \wedge \text{stable}(Q'_2)$$

and therefore as well

$$P_2 \xrightarrow{s} Q'_2$$

We show therefore $(*)$: By $\text{stable}(Q)$ we have $\neg(Q \uparrow)$, therefore $\neg(Q' \uparrow)$. We have that for any Q'' that $\text{stable}(Q'') \vee \exists Q'''. Q'' \xrightarrow{\tau} Q'''$. therefore we obtain a sequence

$$Q_2 = Q'_0 \xrightarrow{\tau} Q'_1 \xrightarrow{\tau} Q'_2 \xrightarrow{\tau} \dots$$

which is either infinite, or ends with a Q'_n which is stable. Since $\neg(Q_2 \uparrow)$, this sequence cannot be infinite. So we obtain a Q'_n such that

$$Q' = Q'_0 \xRightarrow{\langle \rangle} Q'_n \wedge \text{stable}(Q'_n)$$

By R being a DRW-bisimulation and $Q R Q'$, there exists Q'' such that

$$Q \xRightarrow{\tau} Q'' \wedge Q'' R Q'_n$$

by Q stable, we get $Q = Q''$.

We note here that the above argument is classical. When formalising CSP-Agda, which is based on constructive logic, we formalise non-divergence as an inductive data type which expresses that the sequence of τ -transitions is well founded.

The proof of the following lemma is obvious (one easily sees that if R is a bisimulation, $P R P'$ and $P \uparrow$ then $P' \uparrow$):

Lemma 2.4.7 (a) *Any strong bisimulation is a DRW-bisimulation.*

(b) *Any DRW-bisimulation is a weak bisimulation.*

In chapter 10 we will implement strong bisimilarity and divergent-respecting weak (DRW) bisimilarity in CSP-Agda. This will facilitate proofs of algebraic properties for trace, stable failure semantics, and FDI semantics. We will show that strong bisimilarity implies DRW bisimilarity, and both imply equivalence with respect to trace semantics, stable failures semantics and FDI semantics. This allows to carry out more easily proofs of algebraic properties in this semantics by showing first strong or DRW bisimilarity and then obtaining directly equivalence in the other semantics. As an example, we will apply this methodology to prove algebraic laws according to this semantics: we prove commutativity of the interleaving and external choice operators and prove two monadic laws.

2.5 Tool Support

There are three main tools that have been developed and are used to analyse CSP processes. In this section we are going to briefly explain these tools. These tool support checking of CSP syntax, checking of standard properties such as livelock and deadlock, checking of formulas describing properties of CSP and checking of the refinement relation between processes. The first tool is ProBE, the second one is Failures Divergences Refinement (FDR), and the last one is CSP-Prover.

2.5.1 ProBE

ProBE (Formal Systems (Europe) Ltd [2003]) allows the user to investigate the behaviour of CSP processes. The ProBE tool uses as language CSPM

which is machine readable CSP (Roscoe [1998]). ProBE allows the exploration of how a process develops through simulation. It presents how a process proceeds when making different choices, in a tree like structure. It displays part of how a Process develops when making different choices of events. Future processes can be hidden and expanded by clicking on it, or hidden again.

For instance consider the following process

$$\begin{aligned} \text{Sender} &= \text{in}?x : T \longrightarrow \text{mid}!x \longrightarrow \text{ack} \longrightarrow \text{Sender} \\ \text{Receiver} &= \text{mid}?y : T \longrightarrow \text{out}!y \longrightarrow \text{ack} \longrightarrow \text{Receiver} \\ &(\text{Sender} \parallel \text{Receiver}) \setminus (\text{mid}.T \cup \text{ack}) \end{aligned}$$

Part of its exploration through the ProBE tool can be found in Fig. 2.2.

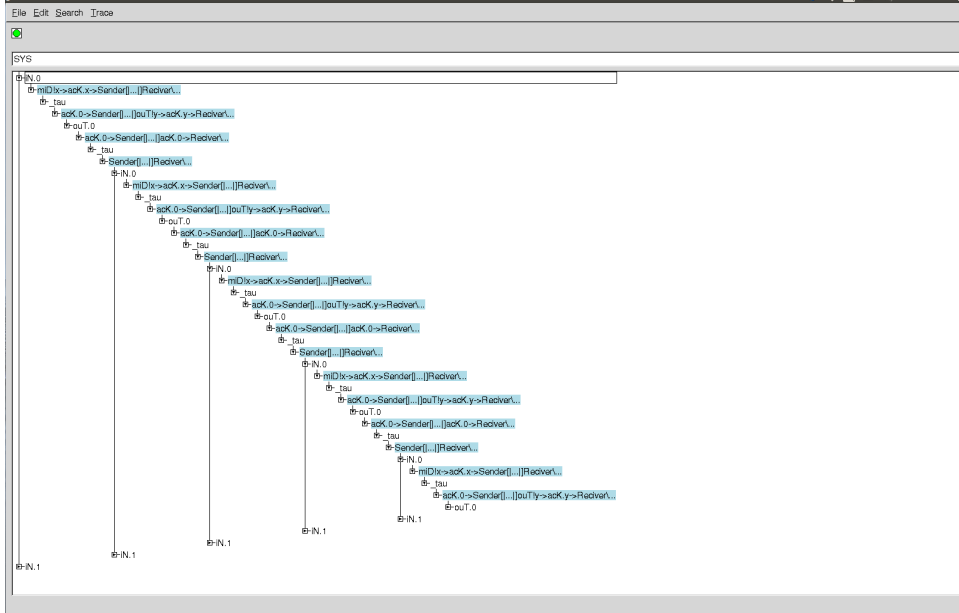


Figure 2.2: ProBE Tool Interface

2.5.2 Failures Divergences Refinement (FDR)

The FDR tool (Failures Divergences Refinement) is the first commercial available tool for CSP. The FDR tool (University of Oxford [2012]) uses as ProBE as input language CSPM. This tool allows the checking of a wide range of correctness conditions, such as livelock, deadlock, and general properties as well as general safety and liveness properties. As an example, consider the example considered in the previous subsection, and the process

$$(\text{Sender} \parallel \text{Receiver}) \setminus (\text{mid}.T \cup \text{ack})$$

Fig. 2.3 shows we can check using the FDR2 tool that the process is free from livelock and deadlock (using a different selection).

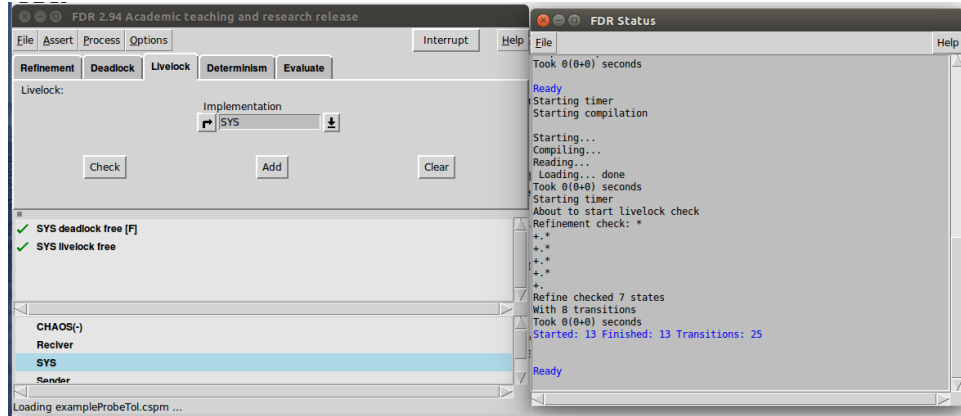


Figure 2.3: FDR2 Tool Interface

2.5.3 CSP-Prover

This tool integrates the CSP theory with the theorem prover Isabelle/HOL where it is used in order to carry out refinement proofs for CSP. CSP-Prover (Isobe and Roggenbach [2005]) is an interactive theorem prover (Nipkow et al. [2002]). The interactive theorem prover Isabelle/HOL is based on ML (Milner [1997]). Whereas the previous tools (See 2.5.2) restrict CSP specifications to finite state systems, the CSP-Prover allows to prove properties of infinite state systems.

Chapter 3

Theorem Prover Agda

Agda is a dependently typed programming language and theorem prover. Types can depend on arbitrary values. This is in contrast to the functional programming languages Haskell and ML, which have are simply typed based on a Hindley-Milner style language (Milner [1978], Damas and Milner [1982], Hindley [1969]), and therefore separate types and values. In this chapter, we give an overview of Agda version 2, see Bove et al. [2009] for a brief overview over Agda 2 and its history. Agda is the latest in a series of dependently typed programming languages Alf (Magnusson and Nordström [1994]), Half (Cederquist [1997]), CHalf (Cederquist et al. [1998]), Alfa (Thomas Hallgren [2017]), Agda 1 (developed by Catarina Coquand, Coquand [2009]) and Agda 2 (Bove et al. [2009], Norell [2009b,a], Agda Community [2017b]).

All of them were extensions of Martin-Löf type theory (Martin-Löf [1984]). See the book (Nordström et al. [1990]) for a more computer science oriented introduction into Martin-Löf type. All of these theorem provers were developed primarily at the University of Chalmers in Gothenburg, Sweden. The current version of Agda is Agda 2, the basis of which was designed and implemented by Ulf Norell in his PhD in 2007 (Norell [2007]). The work in our PhD thesis is carried out in Agda 2. Therefore all references to Agda in this thesis, unless stated explicitly differently, refer to Agda 2. The language of Agda is functional, however using dependent types, with lots of features added over time. Agda supports inductive and coinductive data types. Inductive data types include inductive families. Functions are defined for these types by using termination checked pattern and copattern matching (Abel et al. [2013]). One particular feature of Agda is that it supports induction-recursion. In induction recursion, a set is defined inductively while simultaneously recursively defining a function over it. This allows to define in particular universes, where the inductively defined type is a set of codes, and the recursively defined function maps a code to the type it denotes.

The theorem prover most similar to Agda is Coq (Bertot and Castéran

[2010]). The language of Coq is an extension of the Calculus of Constructions (Coquand and Huet [1988]). Coq allows to define directly inductive data types but no inductive-recursive types.

Both Coq and Agda are proof assistance with dependent types. Logical propositions are identified with certain types using the “propositions as types” principle, based on the Curry-Howard isomorphism (Bertot and Castéran [2010]). To prove a proposition, means to write a program which is an element of the corresponding type.

Most definitions in Agda use pattern matching. Pattern matching is an easy to use syntax to make a case distinction on the constructors of an inductive data type. In Agda we can for instance define an inductive family

$$\mathbf{Vec} : \mathbb{N} \rightarrow \mathbf{Set}$$

where $\mathbf{Vec} \, n$ is the set of n -tuples or vectors of length n .

Then we can define a function

$$f : (n : \mathbb{N})(v : \mathbf{Vec} \, n) \rightarrow B(n, v)$$

which can be defined by making case distinctions on n and v . Coq has a similar mechanism using the “match ... with ...” construct.

Coq distinguishes between the types Prop and Set. Prop allows impredicative quantification. The sort of logical propositions is defined as Prop, whereas data types are elements of Set. This kind of distinction, apart from adding an enormous proof theoretic strength, is useful in order to extract OCaml and Haskell programs from proofs in Coq. The reason is that it allows to separate proofs, which are elements of types in Prop from programs, which are elements of types in Set. Agda does not have this kind of distinction, except for some experimental variant of it (where Prop is still predicative).

Coq is fixed by theoretical work on the calculus of inductive constructions, whereas Agda is more flexible and has over time diverged quite far from its basis in Martin-Löf Type Theory. This allows to add new ideas more easily to Agda.

Coq has the concept of tactics, which allow to automatically generate elements of types. In contrast, in Agda, at the time of writing this thesis, automated generation of elements of types is quite restricted. There are attempts to improve this, but they are still in an experimental stage. Coq allows as well setoid rewrite, which makes writing elements of quotient types, which are represented as setoids, more easily. Both Coq and Agda support type classes and an infinite hierarchy of type universes.

However, the challenge to write proofs by hand in Agda has given rise to new definitions of data types and programs, which can be programmed more easily. This is in particular useful for writing dependently typed programs, where automatically generated code easily becomes inefficient.

In this chapter we will introduce the basic features of Agda and show how they are used in the construction of dependently typed programs. We will introduce as well the syntax of Agda through the use of examples. More information about theorem prover Agda can be found in the Agda Wiki (Agda Community [2017a], Agda Community [2017b]), in Ulf Norell’s PhD thesis (Norell [2007]), and in the book Stump [2016].

3.1 Totality of Agda

Agda is a total language, which means that every program in it must terminate: all computations must terminate and return a value without any run-time error. Without this feature, the logic behind this language becomes inconsistent: For instance, without termination checking, we can define for any type A an element of it by defining it as

$$\begin{aligned} a &: A \\ a &= a \end{aligned}$$

and therefore prove all theorems.

This requires certain checks to be performed, in order to obtain total computations in Agda language: apart from type checking, one needs

- coverage checking, which checks that pattern matching covers all cases
- termination checking, which guarantees that functions are defined by an extension of primitive (co)recursion,
- strict positivity of constructors checking, which makes sure that the (co)data types are strictly positive (otherwise it doesn’t make sense to talk about primitive (co)recursion)

Primitive (co)recursion is now based on the copattern matching, which corresponds to the principle of guarded recursion.

More information about totality can be found in Turner [2004].

3.1.1 Type checking

The procedure that decides if the program conforms to a given set of typing rules is called the type checker. This checker checks that terms are formed

from functions and other operations which are applied in a type correct way. Agda allows to define code having undefined code, called a goal. For a goal it shows the type of the goal, its environment, allows to compute normal forms and types of terms relative to this context. More detailed information about the type checker of Agda can be found in Chapman's PhD thesis Chapman [2009] and in Norell's PhD thesis Norell [2007].

3.1.2 Coverage checking

Because of the use of inductive families and induction recursion, and nested pattern and copattern matching, it can be quite difficult to determine whether a given (co)pattern covers all cases. This check for coverage is performed in Agda by the coverage checker Bove et al. [2009]. A very simple example uses the set of colours defined as follows:

```
data color : Set where
  Red : color
  Green : color
  Blue : color
```

Consider the function `swapColour` which swaps colours `Green` and `Red`: flips it to another one as follows:

```
swapColor : color → color
swapColor Red = Green
swapColor Green = Red
```

This definition has no pattern for `Blue` and if we therefore applied `swapColour` to this value we would obtain a runtime error. It doesn't pass the coverage checker and is rejected by Agda. We note that the coverage checking is not trivial in the presence of inductive families of types. An example is the definition of even numbers and proof that the sum of even numbers is even:

```
data IsEven : ℕ → Set where
  even0 : IsEven 0
  evenSuc : {n : ℕ} → IsEven n → IsEven (succ (succ n))

even+ : (n m : ℕ) → IsEven n → IsEven m → IsEven (n + m)
even+ .0 m even0 pm = pm
```

```
even+ .(succ (succ n)) m (evenSuc {n} pn) pm = evenSuc (even+ n m pn pm)
```

Pattern matching on the proof of `lsEven n` and matching it with `even0` forces the argument `n` to become `0`, where the dot indicates that this is forced. Similar in the second line the first natural number is forced to be `.(succ (succ n))`.

3.1.3 Termination checking

As mentioned before, Agda is a total language, meaning that each program must terminate, since Agda without termination checker makes the logic behind this language inconsistent. In most functional programming languages recursion can be used freely; for instance, the partial function `div` can be defined in Haskell as follows:

$$\text{div } m \ n = \text{if } (m < n) \text{ then } 0 \text{ else } 1 + \text{div } (m - n) \ n$$

When we use this definition in Agda, it is rejected, and must be rejected, since this function is partial: if `n` is 0 it will not terminate. Since the termination problem is undecidable, the termination checker of Agda cannot accept exactly all terminating programs, but only a subset of them. It does so by accepting structurally recursive programs, which is an extended form of primitive recursion. So even after fixing the above definition of `div` it would in this form still be rejected by Agda, since it is not an instance of structural recursion.

Primitive recursion

In order to make sure that all functions terminate, one solution is to force all recursions to be instances of extended primitive recursion. This approach was taken in Martin-Löf type theory (Martin-Löf [1984]) where all recursions need to be an instance of extended primitive recursion. Primitive recursion is a special case of structural recursion on a well-founded data type. The main principle of primitive recursion is that recursive calls are made on structurally smaller arguments, which guarantees termination.

For instance, consider the type of natural numbers:

```
data N : Set where
  zero : N
  succ : N → N
```

This means that we have a new type $\mathbb{N} : \text{Set}$ with operations $\text{zero} : \mathbb{N}$ and $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$. Furthermore, the elements of \mathbb{N} are those constructed from applying these operations. Therefore functions can be defined by case distinction of these operators using pattern matching, for example:

```
double :  $\mathbb{N} \rightarrow \mathbb{N}$ 
double zero = zero
double (succ n) = succ (double n)
```

Nested patterns and mutual definitions are allowed, for example:

```
mutual
  f :  $\mathbb{N} \rightarrow \mathbb{N}$ 
  f zero          = 1
  f (succ zero)   = 2
  f (succ (succ x)) = g x

  g :  $\mathbb{N} \rightarrow \mathbb{N}$ 
  g zero = 3
  g (succ n) = n
```

The coverage checker checks completeness, whereas the termination checker ensures that the recursive calls follow a schema of extended primitive recursion.

We can define addition of a natural number as follows:

```
_+_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
zero  + m = m
succ n + m = succ (n + m)
```

Here termination checking will succeed, since in the second line, the first argument becomes smaller in recursive call (n is structurally smaller than $\text{succ } n$; in fact, since we are using natural numbers it is actually a smaller number). Working with this kind of recursion is often inconvenient in practice since we deal with only one argument at a time. More details can be found in the article Martin-Löf [1984] and in the article Dybjer [1994] on inductive families.

Structural recursion

In some programming language, primitive recursion is the only recursion allowed, as for instance, in Coq. This is not a problem since Coq can define more complex functions using its tactics. Agda allows more general forms structural recursion in which recursively calls on subexpression of the argument are allowed. For instance, the Fibonacci number can be defined as follows by using nested pattern matching:

```
fib : ℕ → ℕ
fib zero = zero
fib (succ zero) = succ zero
fib (succ (succ n)) = fib n + fib (succ n)
```

3.1.4 Equality

Agda has three kind of equalities equalities. One is definitional equality, which is a decidable equality used during type checking. It is essentially based on that two terms are equal if they have up to α - and η -equality have the same normal form. The second one is propositional equality, which is a user defined equality. An example in this thesis will be the definition of bisimilarity on processes which is defined as a binary predicate on processes. Another example would be to define equality on the function type $A \rightarrow B$ extensionality by defining that two functions are equal if for equal elements of A they return equal elements in B . This allows to define undecidable equalities on types, which can be used for formal specifications. The third equality is the intensional equality type. It is the least proposition closed under reflexivity. Since for type checking purposes definitionally equal terms are identified, it contains definitional equality. Intensional equality is relatively weak and often needs to be replaced by a propositional equality. in order to prove theorems. We will introduce intensional equality below in Subsect. 3.2.12.

3.2 Types and Expressions In Agda

Agda has an infinite sequence of type levels. The lowest type level is for historic reasons called Set. Types in Agda are given as inductive types, coinductive types, dependent function types, Size type, record types, and a generalisation of inductive-recursive definitions and inductive-inductive definitions.

3.2.1 Inductive Data Types

Mathematical induction is the backbone of programming and program verification (Leino and Moskal [2014]). Many data structures can be defined as inductive data types, also called algebraic data types (Bird and Wadler [1988]). Inductive data type allow to introduce elements of them by using constructors, and using them by recursion (Beckert et al. [2007]), and proving properties by induction (Sheeran et al. [2000]). The elements of an inductive data type in theorem proving are well-founded, which means the element can be considered as a tree with no infinite branches. If there is finite branching, the objects are by König's Lemma finite. Inductive data types in Agda are dependent versions of algebraic data types as they occur in functional programming. There are several levels of types, the lowest (for historic reasons) being called **Set**. The inductive data type is given as a set A with constructors which are strictly positive in A^1 . As an example are the set of natural numbers, which we for convenience repeat here:

```
data N : Set where
  zero : N
  succ  : N → N
```

Another example of an inductive data type is the definition of Maybe, which adds to the elements of A , written as (**just** a) one extra element **nothing**. It is a degenerated inductive data type, because it is not a recursive definition:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A
```

Agda allows as well simultaneous inductive definitions. We define her the collection of finite sets (**Fin** n) having n elements. For $n \geq 1$ we have that **zero** is an element of (**Fin** n), and if m is an element of (**Fin** n), then (**suc** m) is an element of (**Fin** (**succ** n)). One easily sees that with this definition (**Fin** n) has exactly n elements. The definition in Agda is as follows:

```
data Fin : N → Set where
  zero : {n : N} → Fin (succ n)
  suc  : {n : N} → Fin n → Fin (succ n)
```

¹Which is explained in Sect. 3.2.2

3.2.2 General Form of Inductive Data Types

The general form of inductive data types in Agda is as follows:

```
data D : Set where
  C1 : A1
  C2 : A2
  .
  .
  .
  Cn : An
```

Assume each A_i has the form

$$(y_1 : B_1) \rightarrow \dots (y_n : B_n) \rightarrow D$$

where the argument type B_i is either non-inductive (and does not mention D at all) or inductive, in which case it has the form:

$$(z_1 : C_1) \rightarrow \dots (z_k : C_k) \rightarrow D$$

where D must not occur in any C_j .

For instance the following definition of `Bad` is not accepted by the positivity checker of Agda:

```
data Bad : Set where
  bad : ( Bad1 → Bad2 ) → Bad3
```

`Bad` is in position 1 in negative position; in position 2 and 3, it will be accepted.

3.2.3 Dependent function type

The dependent function type is like a simple function type but the result type depends on the value of the argument. As an example, we write in Agda $(n : X) \rightarrow Y$ for the type of functions taking an argument n of type X and returning a result of type Y , where n may appear in Y . For example, in Agda we can define

```
id1 : (A : Set) → A → A
id1 A x = x
```

The above is a dependent function taking a type argument A and an element of A and returns that element.



3.2.4 Guardedness checking in coinductive programs

The design of programming languages using infinite coinductive types is challenging, especially if termination needs to be guaranteed.

In comparison, most programmers feel comfortable with inductive data types, whereas coinductive data type are considered to be more complex. Primitive corecursion or Guarded recursion is a principle for defining termination checked programs. This principles is built into the languages Agda and Coq.

When using coinductive types defined by their introduction rules, guarded corecursion is the principle of recursion which allows arbitrary recursive calls as long as they are guarded by a coinductive constructor. Productivity is ensured by guardedness. For more information about the coinduction approach, see the articles by Danielsson and Altenkirch [2009], Altenkirch and Danielsson [2010] and Danielsson [2010]. In the newer approach in Agda coinductive types are defined by their elimination rules (Abel et al. [2013], Abel and Pientka [2013], Setzer et al. [2014]). Primitive corecursion is there defined differently. An element of a coinductive type can be defined by copattern matching on its eliminators (also called observations). This definition can make a corecursive call, provided no other function is applied to it. For more details see for instance Setzer [2016].

3.2.5 Coinductive Data Types

There are two ways of defining in Agda coinductive types. The older one defines coinductive types by their introduction rules. One adds first the following special built-in functions (called “musical notation”) which have the following signatures:

$$\begin{aligned} \infty & : \forall \{A\}. A \rightarrow \text{Set} \\ \#_ - & : \forall \{A\}. A \rightarrow \infty A \\ \flat & : \forall \{A\}. \infty A \rightarrow A \end{aligned}$$

The idea is that coinductive arguments of a coinductively defined set A are denoted by ∞A . The operations $\#_ -$ (delay) and \flat (force) are used to convert between A and ∞A . The objects created by corecursion are defined using these functions, and destructured with coinduction.

More generally mixed inductive-coinductive types can be defined, see for instance Danielsson and Altenkirch [2010]. In order to define a type A in the coinductive way we denote any inductive arguments by A and any coinductive arguments by ∞A . ∞A stands for the type of delay computation of A . For example, we can define stream as a coinductive type in Agda as follows:



```
data stream : Set where
  _::_ : ℕ → ∞ stream → stream
```

Here the second argument of `_::_` is `∞ stream` and therefore a coinductive argument. We can define elements of `stream` by using corecursion. An example of a infinite sequence of zeros as a stream is defined as follows:

```
stream0 : stream
stream0 = zero :: (# stream0)
```

The value which constructed by corecursion does not need to terminate but has to be productive. We don't expand a corecursive definition infinitely, but only lazily as far as needed. Guardedness means that a recursive definition is guarded by at least one constructor, and that no other operations are applied to a corecursive definition.

For example the set of increasing streams `stream` starting with n is defined as follows:

```
inc : ℕ → stream
inc n = n :: (# inc (succ n))
```

We can use case distinction in order to destruct a `stream`. For instance, in the following function, adds two streams together pointwise. The definition is by coinduction on the constructed stream:

```
addStream : stream → stream → stream
addStream (x :: x1) (y :: y1) = (x + y) :: (# addStream (b x1) (b y1))
```

The second approach to representing coalgebras in Agda, which we use in this thesis, is the approach of defining coinductive types in Agda by their elimination rules as introduced in Abel et al. [2013], Setzer et al. [2014]. The standard example is the set of streams:

```
record Stream (i : Size) : Set where
  coinductive
  field
    head : ℕ
    tail : {j : Size < i} → Stream j
```

If we first ignore the arguments `Size`, `Size<` we see that the type `Stream` is given as a record type in Agda. It is defined coinductively by its observations `head`, `tail`.

Elements of `Stream` are defined by copattern matching, i.e. by determining the result of applying `head`, `tail` to them. A simple (non-recursive) operation is the function `cons` for adding a new element in front of a stream:

```
cons : ∀ {i} → ℕ → Stream i → Stream (↑ i)
head (cons n s) = n
tail (cons n s) = s
```

Without sizes, in recursive definitions only recursive calls to the function being defined are possible. Especially, no functions can be applied to the recursive calls. However, there are no restrictions on the arguments, the recursive function calls can be applied to. This restriction on the recursive definitions is called the principle of guarded recursion Coquand [1994] or primitive corecursion. As an example, we give the pointwise addition of two streams:

```
_+s_ : ∀ {i} → Stream i → Stream i → Stream i
head (s +s s') = head s + head s'
tail (s +s s') = tail s +s tail s'
```

`_+s_` makes a recursive call to `tail s +s tail s'`. Note that `s`, `s'` are arguments of `_+s_`, so we can apply `tail` to them freely.

Without the restriction of guarded recursion, one could define non-productive definitions, e.g. define `tail (f x) = tail (f x)`. However, the guardedness restriction makes it difficult to define streams in a modular way, since we cannot in a recursive call refer to other functions for forming streams at all, although many operations will not cause problems. Therefore Abel has introduced sized types Abel [2006, 2016] in the context of coinductive types, which allows to apply size preserving and size increasing functions to recursive calls.

Sizes are essentially ordinals (without infinite branching one can think of them as natural numbers); however, there is an additional infinite size ∞ . We have as operations for forming sizes the infinite size ∞ , the successor operation on sizes \uparrow , and have the type of sizes less then i denoted by `Size< i`.

For ordinal sizes $i \neq \infty$, a stream $s : \text{Stream } i$ allows up to i times of applications of `tail`. The true streams is the set `Stream ∞` and $s : \text{Stream } \infty$ allows arbitrary many applications of `tail`. When defining an element $f : (i : \text{Size}) \rightarrow A \ i \rightarrow \text{Stream } i$ by corecursion, $(\text{tail } (f \ i \ a) \ \{j\})$ must be an

element of size $\geq j$ which can refer to a recursive call $(f\ j\ a')$, and we can apply functions to it as long as the resulting size is $\geq j$. Elimination on the recursive call is prevented since we don't have access to any size $< j$. However, we can apply size preserving and size increasing functions to the recursive call. This guarantees that streams are productive. We have $\infty : \text{Size} < \infty$, so a recursive definition of elements of $\text{Stream } \infty$ can refer to itself.

An example is the delay monad, which has been formulated by Abel and Chapman [2014]:

```
mutual
  record  $\infty\text{Delay}$  (i : Size) (A : Set) : Set where
    coinductive
    field
      force : {j : Size < i}  $\rightarrow$  Delay j A

  data Delay (i : Size) (A : Set) : Set where
    now : A  $\rightarrow$  Delay i A
    later :  $\infty\text{Delay}$  i A  $\rightarrow$  Delay i A
```

The delay monad represents partial elements of a data type: they can be defined elements given by `now a` or they can be elements `later a` which are computed later. Infinite sequences of `later` are allowed, in which case an element is never defined, i.e. undefined.

The `Delay` type is represented using the `mutual` definition of a coinductive record and an inductive data type. A single observation `force` is used in order to interact with the coalgebra `∞Delay` given as a `record` type. When forced, elements of `Delay` are obtained. We can make pattern matching on this time to see if the value available is `now` or `later`. If the value is available `later` then an element of `∞Delay` is obtained, and we can `force` it again. In the definition above, `Size` is considered as an index type for `Delay` and `∞Delay` , understood as *observation depth*.

3.2.6 Induction-Recursion

In intuitionistic type theory, inductive-recursion is considered to be powerful definition method. The origin of this idea appeared for the first time in Martin-Löf definition of *universes à la Tarski* Martin-Löf [1982, 1984]. Later Peter Dybjer abstracted from these examples and introduced the concept of induction-recursion Dybjer [1991, 1992, 2000], Dybjer and Setzer [2003]. By a *universe* we mean a type the elements of which represent a type. In the à la Tarski version of universes the elements of the universe are codes and

there is a decoding function which maps the code to the type it denotes.

The universe is defined inductively while the decoding function is defined simultaneously recursively. For instance we can define the type \mathbf{U} and the function \mathbf{T} simultaneously by induction-recursion: We give here the code expressing that \mathbf{U} is closed under \perp , \top , \mathbf{Bool} , and Π . Closure under other sets needed can be easily added:

```
mutual
  data U : Set where
    ⊥'    : U
    ⊤'    : U
    Bool' : U
    Π'    : (a : U)(b : T0 a → U) → U

  T0 : U → Set
  T0 ⊥'    = ⊥
  T0 ⊤'    = ⊤
  T0 Bool' = Bool
  T0 (Π' a b) = (x : T0 a) → T0 (b x)
```

The false proposition, \perp , is defined as the empty data type having no constructors. It is the false formula, since it has no proof. The true proposition, \top , is defined as a datatype with a single element, so it has exactly one proof. These propositions are defined in Agda as follows:

```
data ⊥ : Set where
```

```
data ⊤ : Set where
  triv : ⊤
```

These propositions can be used to translate Boolean values into propositions:

```
⊤ : Bool → Set
⊤ true = ⊤
⊤ false = ⊥
```

$\top b$ is the proposition expressing that b equals true. The type of Booleans is defined in Agda as follow:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

These definitions will be used in Chapter 5 in order to define the type of choicesets.

The set `Bool` together with `T` is an example of a small *universe*. A universe consists of a set `U` of codes for the elements of the universe and a decoding function $T : U \rightarrow \text{Set}$, which maps each element `U` to the set it denotes. This is a way of defining a set of sets as a set. In this case the universe `Bool` has elements `true` denoting the true formula, which is therefore mapped by `T` to the true formula, and `false` denoting the false formula mapped by `T` to the false formula. Here, the set of codes and the decoding function can be defined separately, in most examples in dependent type theory, both need to be defined simultaneously by an inductive-recursive definition. In Chapter 5 we will define universe consisting of `Choice` together with `ChoiceSet`, which will be defined inductive-recursive.

3.2.7 Records

Record types are used in order to describe the grouping of several types into one type. There are many different notations for records in programming languages. For instance, the record in C is denoted by the keyword `struct` and defined as follows:

```
struct data
  {int year ;
   int month ;
   int day ;
  } ;
```

Record types are defined in Agda as indicated by the following example:

```
record AB : Set where
  constructor pair
  field
    a : ℕ
    b : Bool
```

The above example defines a new record type `AB` with two fields. The first field is `a`, which has type `ℕ` and the second field is `b`, with type `Bool`. In this

definition of `AB` the constructor is `pair`. In addition Agda allows dependent record type where the type of one field depends on other fields.

Elements of a `record` type can be defined in three ways: By using a `constructor`, by using the record syntax, or by defining them using elimination rules.

In order to introduce an element of a record type by a constructor, one needs to add to the definition of the record type a constructor, using keyword `constructor`, as in the previous example. This part of the definition is optional. This allows to define elements of a `record` type in a more readable and clean way. For instance, the element of `AB` record can be defined, assuming `x : ℕ` and `y : Bool` as follows:

```
n = pair x y
```

The Agda record definition of an element is always available. The element of `AB` above can be defined as follows:

```
n = record { a = x ; b = y }
```

The third way is by elimination rules

```
a n = x
b n = y
```

An example of a record type is the Σ -type. If `A : Set` and `B : A → Set`, then $\Sigma A B$ is the set of pairs (a, b) where $a : A$ and $b : B a$. We introduce here a definition which is generic in the set levels:

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

We use as frequently the non-dependent product. Although it can be reduced to the Σ -type, it behaves often better because the types can be inferred for it, whereas because of the second argument of the Σ -type is a function type, it usually cannot be inferred:

```
data _×_ (a b : Set) : Set where
  _,_ : a → b → a × b
```

3.2.8 Mixfix Operators and Unicode

Agda has a mechanism for defining infix and mix-fix operators, where the arguments of infix and mix-fix operator are denoted by the underscore (`_`). For example, disjunction which is infix on truth value can be defined as follows:

```
_or_ : Bool → Bool → Bool
false or m = m
true or m = true
```

In order to declare precedence for the operator, Agda has the reserved key words `infixr`, `infixl` and `infix`, for example:

```
infixl 10 _or_
infixl 11 _and_
infixl 12 if_then_else_
```

The priority of the operator defined by the number near the reserved word, for instance, `_and_` binds more than `_or_` since the number associated with it is higher. `infixl` denotes left associative operation; for example `(a or b or c)` is parsed as `((a or b) or c)`. If we had defined it as `infixr` it would be parsed as `(a or (b or c))`. With `infix` this definition would cause a parsing error, since it cannot be parsed in a unique way. For more details, see article Danielsson and Norell [2011]. Agda supports Unicode symbols by default whereas in Coq and Haskell they are only available as an extension. The use of Unicode feature allows to write more readable code, for instance we can use the symbols Σ , \forall and \exists . Mixfix and Unicode feature allow to write in Agda code which looks very similar to what one would write down by hand. For example, the type of Booleans and the disjunction operation can be defined as follows:

```
data  $\mathbb{B}$  : Set where
  true  :  $\mathbb{B}$ 
  false :  $\mathbb{B}$ 
```

```
_ $\vee$ _ :  $\mathbb{B}$  →  $\mathbb{B}$  →  $\mathbb{B}$ 
true   $\vee$  true  = true
true   $\vee$  false = true
false  $\vee$  true  = true
```

```
false ∨ false = false
```

3.2.9 Implicit Arguments

Agda offers a mechanism to mark arguments as implicit. Implicit arguments can be omitted, in case the type checker can uniquely determine the omitted argument. This allows to hide unnecessary argument, and therefore to simplify code, make it more understandable and more readable.

Implicit arguments are marked by curly brackets ($\{\}$). For instance, we can define:

```
id : {A : Set} → A → A
id x = x
```

In the above function `id`, the argument $A : \text{Set}$ is implicit. The next definition is the same as the following definition, where the argument $A : \text{Set}$ is explicit:

```
id1 : (A : Set) → A → A
id1 A x = x
```

Using the first definition, we have to hide the argument, for instance:

```
True : Bool
True = id true
```

```
Zero : ℕ
Zero = id zero
```

With the second definition, we have to state the argument explicitly:

```
Zero1 : ℕ
Zero1 = id1 ℕ zero
```

```
true1 : Bool
true1 = id1 Bool true
```

In order to make the implicit argument explicit, curly brackets can be used, for instance:

```
id {ℕ} zero : ℕ
```

```
id {-} zero : ℕ ..... (*)
```

In (*) the underscore (-) indicates that the type checker has to figure out what the type of argument should be. If the type checker fails to infer the value, then it will issue an error message.

3.2.10 Module System

The module system in Agda is considered as a mechanism that is intended to structure Agda code by separating it into different modules which might occur in different files. This feature allows separate compilation and allows as well the use of parametrised modules. It is also useful for structuring larger developments. Modules are introduced in Agda using the keyword `module`. A module from a different file is imported into the current file using the keyword `import`, e.g. `import maybe`. After `import Nat`, names from the module `Nat` can be used using a qualified name, for instance:

```
import maybe
```

```
fMaybe : {A B : Set} → (A → B) → maybe.Maybe A → maybe.Maybe B
fMaybe f maybe.nothing = maybe.nothing
fMaybe f (maybe.just x) = maybe.just (f x)
```

The second reserved word is `open Module`, which brings everything from the module into scope, i.e. no qualified name is needed anymore. An example is

```
import Nat
open Nat
```

```
Z : ℕ
Z = zero
```

We can combine `open` and `import` into one step by using `open import`, for instance:

```
open import maybe
```

```
fMaybe : {A B : Set} → (A → B) → Maybe A → Maybe B
fMaybe f nothing = nothing
```

```
fMaybe f (just x) = just (f x)
```

Agda allows as well to control the names brought into scope by deciding explicitly which names to open (keyword `using`), or by hiding names (keyword `hiding`) and by renaming names (keyword `renaming`).

3.2.11 Postulated Types

Agda allows, using the keyword `postulate`, to postulate a type or function, where the constant of this type is introduced without any reduction rule. For instance, we can postulate a type and function as follows:

```
postulate A : Set
postulate a' : A
postulate _==_ : A → A → Set
postulate _>'_ : A → A → Set
```

This mechanism assumes that certain constructions exist, without defining them. It even allows to define an element in the empty set. Therefore a proof is only correct, if no postulates occur. In the above example we introduce a set `A` and introduce an element `a'` of this set; we know nothing else about `A` and the binary relation `_==_`.

3.2.12 Let, Where, With-expressions, Mutual Definitions, Intensional Equality, and Rewrite

Agda offers `let` and `where`—expressions in order to declare local definitions. In comparison, `where`—expressions allow a pattern matching or recursive function, whereas pattern matching and recursive functions are not allowed in `let`—expressions. In Agda `let`—expressions are represented as follows:

```
let
  a1 : A1
  a1 = s1
  a2 : A2
  a2 = s2
  ...
  an : An
  an = sn
in t
```

In the above definition, the **let**-expressions introduce the following new local constants:

$$\begin{aligned} a_1 & : A_1 \text{ s.t. } a_1 = s_1, \\ a_2 & : A_2 \text{ s.t. } a_2 = s_2, \\ & \dots \\ a_n & : A_n \text{ s.t. } a_n = s_n \end{aligned}$$

On the other hand, **where**-expressions allows pattern matching recursive definitions. An example of the use of primitive recursion and pattern matching in **where**-expressions is as follows:

```
revList : (A : Set) → List A → List A
revList A list = refAux list []
  where
    refAux : List A → List A → List A
    refAux []      xy = xy
    refAux (x :: xs) xy = refAux xs (x :: xy)
```

The reserved word **mutual** is used in Agda in order to simultaneously define two or more functions or data types that depend on each other, for instance:

```
mutual
  data Even : Set where
    zero : Even
    suc : Odd → Even

  data Odd : Set where
    suc : Even → Odd
```

Alternatively, one can omit **mutual** by first defining the signatures of the types used and then giving their definitions. The example above can be rewritten as follows:

```
data Even : Set
data Odd  : Set

data Even where
  zero : Even
```



```
suc : Odd → Even
```

```
data Odd where
  suc : Even → Odd
```

In this example the signature of `Odd` is

```
data Odd : Set
```

(without keyword `where`) and `: Set` is omitted in the definition.

In Agda we use the construct `with` in order to abstract function f over the value of an auxiliary expression e . This adds another argument to f , over which we can then pattern match. An example is as follows: instance:

```
MinN : ℕ → ℕ → ℕ
MinN x y with (x < y)
MinN x y | true = x
MinN x y | false = y
```

Here, the equations of `MinN` have an extra argument, separated from the original one by a vertical bar. In this example, we did not carry out any further pattern matching on x and y , the only case distinction we did was on $(x < y)$. Repeating the left-hand side is tedious so that we can replace it with `...|`. So we can define `MinN` as well as follows:

```
MinN : ℕ → ℕ → ℕ
MinN x y with (x < y)
...| true = x
...| false = y
```

Intensional equality on a set A , which was mentioned in Subsect. 3.1.4, is defined as an indexed inductive definition, which has a proof `refl` of type $x \equiv x$:

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

The `rewrite` construct takes as argument a proof of a propositional equality, and rewrites the goal and the context of the given equation by using this equation. An example is as follows:



```

+0 : ∀ n → n + zero ≡ n
+0 zero = refl
+0 (succ n) rewrite +0 n = refl

```

In this example, in the `(succ n)` case, by using the equation `(+0 n)` from the induction hypothesis, we have the additional rewrite rule that `(n + zero)` rewrites to `n`. Therefore the goal `suc n + zero ≡ n`, which by normal rewrite rules rewrites to `succ n + zero ≡ succ n`, is rewritten to `succ n ≡ succ n`, and we can use `refl` to prove this goal.

The rewrite construct can be reduced to the `with` construct. Our example can be reduced to the following code:

```

+0' : ∀ n → n + zero ≡ n
+0' zero = refl
+0' (succ n) with n + zero | +0' n
+0' (succ n) | .n | refl = refl

```

3.2.13 BUILTIN and Primitive

Agda uses keywords `BUILTIN` and `primitive` in order to use efficient native implementations of specific inductive types. The most common example in Agda is the natural number data type. In the following example we illustrate the idea of this mechanism:

```

data N : Set where
  zero : N
  succ : N → N

```

```

{-# BUILTIN NATURAL N #-}

```

This means that the `BUILTIN` “NATURAL” is used for `N`. Agda will instead of dealing with natural numbers as constructed by the constructors `succ` and `zero` use Haskell’s native natural numbers. It allows to write as well decimal numbers as elements of `N`.

Agda will check that `N` is an inductive construction with type `Set`, which has two constructors, which have the type of `zero` and `succ` above.

Using these declarations we can write `24 : N` instead of

$$\underbrace{\text{succ} (\text{succ} (\text{succ} (\text{succ} (\dots (\text{succ zero})\dots)))\dots)}_{24}$$

It allows as well in patterns to write `0` for the constructor `zero` : `N`.

The second main example of `BUILTIN` is `List`. It is used in Agda as follows:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

The Agda construct `primitive` is similar to `BUILTIN`, but for types which are represented in native Agda as `postulate`, i.e. they do not β -reduce for open terms and are not implemented in Agda. This means, they behaves as a black box which is not checked.

3.3 Setup of Agda

Agda offers an interactive interface, in order to facilitate writing of code. It allows as well to write compiled code.

3.3.1 Interactive Interface

The interactive interface of Agda is based on Emacs. Without this interface writing code with dependent types would be complex. The interface allows to interactively refine code. When code is loaded (i.e. type checked), the code will be partly highlighted either in green, red, and in yellow. A code highlighted in red means non-termination; code highlighted in yellow means that implicit arguments are not inferable. Goals are parts of the code which have not been written yet, and are highlighted in green. Agda has a special goal menu, which allows for goals to determine the type needed, the context, to evaluate in its context terms to normal features, to solve it automatically, to refine the goal, and many more features.

The Agda mode of Emacs offers many short-cut keys: for instance, for loading a file, compiling a file, killing and restarting Agda, showing goals, moving to the next and previous goal, and as well short cuts for the goal menus.

3.3.2 Compiled Version of Agda

The mechanism used in order to create executable code of Agda is by compiling it into Haskell. Different compilers can be used, the main one being

the Glasgow Haskell compiler GHC (Peyton Jones et al. [1993]). Agda uses the key word **COMPILE** in order to replace Agda definitions by corresponding Haskell definitions, followed by the compiler (e.g. GHC) and the definition. It has as well a key word **FOREIGN** to carry out definitions needed for **COMPILE** directive. Consider for instance the definition of List:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

In order to bind the **List** data type in Agda to the **List** data type in Haskell, we first define the type of Agda Lists and then define the List using this type:

```
{-# FOREIGN GHC type AgdaList a = [a] #-}
{-# COMPILE GHC List = data AgdaList ([] | (:)) #-}
```

Another example is the **IO** monad. It is a postulated type in Agda, which is compiled as the Haskell IO monad:

```
postulate IO : Set → Set
{-# COMPILE GHC IO = type IO #-}
```

In case of String we can use the built-in type of strings, but translate string commands using it into corresponding interactive programs of Haskell:

```
postulate String : Set
{-# BUILTIN STRING String #-}

postulate putStrLn : String → IO Unit
{-# COMPILE GHC putStrLn = (\ s -> putStrLn (Data.Text.unpack s)) #-}
```

More information about compiler mechanisms in Agda can be found in the MAlonzo article Benke [2007]) and in Hausmann et al. [2015].

3.3.3 Interactive Programs in Agda

Interactive programs can be written in Agda using the HS-monad, which is a dependently typed version of the IO monad, and which was developed in Hancock and Setzer [2000a,b, 2005], Setzer and Hancock [2004]. The theoretical basis for the IO monad was developed by Moggi Moggi [1991]. It was pioneered by Peyton-Jones and Wadler Wadler [1990, 1995, 1997, 1998],

Peyton Jones and Wadler [1993] in order to represent interactive programs in functional programming, especially in Haskell.

The main idea of the IO monad is that that an interactive program is based on an IO interface, which introduces a set of commands which can be executed in the real world, together with a set of responses the real world can return in response to such a command. An IO program iteratively issues commands from his interface and continues depending on the response from the real world. The IO data type is a coinductive form of a Peterson-Synek tree Petersson and Synek [1989], except that they also have the option to terminate and return a value. This allows for monadic composition of programs, that is, sequencing one program with another program, where the second program depends on the return value of the first program. It is coinductive, since interactive programs can potentially run for ever.

The interface of an IO program is given by a set of real world commands `Command` and a set of responses the real world can return in response to a command executed:

```
record IOInterface : Set1 where
  field
    Command : Set
    Response : Command → Set
```

The set of interactive programs coinductively as a set `IO`, which by using eliminator `force` eliminates into the set `IO'` of IO shapes. The shapes of `IO'` are `(return' a)` for a program which terminates returning value a , and `(do' c f)` for a program which executes in the real world command c , and depending on the response $r : I.`Response` c the world returns continues as an interactive program.$

Monadic composition (`>>=`) of programs allows to combine an IO Program with another program which depends on the result of the first one. It is defined as follows (depending on an interface I ; we define as well a version `>>`, in which the second program does not depend on the first one):

```
_>>=_ : ∀{i}{A B : Set}(m : IO' i I A) (k : A → IO (↑ i) I B) → IO' i I B
do' c f >>= k = do' c λ x → f x >>= k
return' a >>= k = (k a) .force
```

```
_>>=_ : ∀{i}{A B : Set}(m : IO i I A) (k : A → IO i I B) → IO i I B
(m >>= k) .force = m .force >>= k
```

```
_>>_ : ∀{i}{B : Set} (m : IO i I ⊤) (k : IO i I B) → IO i I B
```


$$m \gg k = m \gg= \lambda _ \rightarrow k$$

In this thesis the only IO interface we use is a console interface `console`. It has two commands. The first command (`putStrLn str`) prints a string `str` on the console. The world only returns that it has performed it, so its an element of the one element set `⊤`. The second command `getLine` asks for a string entered by the user. The world response with the String which was input by the user, and depending on it the program continues. The complete definition of `IOconsole` is as follows:

```
data ConsoleCommand : Set where
  putStrLn : String → ConsoleCommand
  getLine   : ConsoleCommand

ConsoleResponse : ConsoleCommand → Set
ConsoleResponse (putStrLn s) = ⊤
ConsoleResponse getLine      = String

console : IOInterface
console .Command = ConsoleCommand
console .Response = ConsoleResponse

IOConsole : Size → Set → Set
IOConsole i = IO i console
```

There exist a library (Abel et al. [2016]) which supports the writing of interactive programs and of object based programs in Agda. See as well the article Abel et al. [2017], which describes this library in great detail.

Elements of `IO` are compiled via the `COMPILE` directive into native IO programs of Haskell which are than compiled into native code and executed.

Just before submitting the final version of this thesis, Haskell's `do` notation has been ported to Agda, and the keyword `do` can no longer be used as a constructor. It would be possible to use `do'`, but we use as well `do` for defining elements of `IO` directly. Therefore, this constructor will be replaced by `exec` in future instance of CSP-Agda.

Chapter 4

Review of Literature

In this chapter, an overview of related work presented in seven parts. First, in Sect. 4.1 we provide an overview of using formal methods in an industrial environment. Then, in Sect. 4.1 we present related work regarding using process algebra modelling and verification of industrial-strength systems. In Subsect. 4.2.1 we present related work on verifying software in CSP. In Sect. 4.3 we give an overview of theorem provers. In the next section 4.4, we start to investigate work on using functional programming to define processes algebras. In the next section 4.5 we present work on defining process algebras in dependent type theory. Then (Sect. 4.6) work on defining process algebras in a coalgebraic manner is presented. In the last section 4.7, we investigate the work on using the theorem prover Agda as a platform for modelling and verifying systems.

4.1 Formal methods

Failure in discovering an error in a critical safety system can lead to a catastrophic situation. Therefore, in the field of industrial safety critical systems, the verification step is considered as crucial. Formal verification and validation is a vital step for the certification of many critical systems, e.g. railway systems, see Cimatti et al. [2012].

Formal methods are mathematical techniques for developing and verifying software and hardware systems, usually assisted by tools. The use of mathematically rigorous techniques allows users to analyse and verify systems at any stage of the program life cycle: requirements, specification, architecture, design, implementation, testing, maintenance, and evolution (Woodcock et al. [2009]).

The requirements stage is considered as an essential step in a high-quality software development process. Easterbrook et al. [1998] successfully present three case studies of using formal methods in requirements modelling.

In the specification level, this approach plays a dominant role in specifying software. Here, the behaviour of sequential systems can be described using certain formal methods, for instance, Z (Spivey [1988]), VDM (Jones [1990]), and Larch (Garland et al. [1993]).

In order to specify systems in a concurrent manner, other methods such as CSP (Hoare [1978]), CCS (Milner [1982]), state charts (Harel [1987]), temporal logic (Pnueli [1977], Manna and Pnueli [2012], Lamport [1994]), and I/O automata (Lynch and Tuttle [1987]) are used.

Complex software systems need a rigorous organisation of the architectural structure of their segments: a model of the system that suppresses the implementation specification is needed, in order to enable the designer to focus on the analyses and decisions that are essential to structuring the system in order to meet its requirements (Allen [1997], Lamsweerde [2003]).

The formal architectural description language WRIGHT provides a practical basis for a formalisation of the abstract behaviour of architectural components and connectors (Allen [1997]).

Formal methods are used in software design. Refinement is considered as a general approach for adding details to a software design in incremental steps (Mo [2005]). Data refinement plays a central role in methods such as VDM (Jones [1990]) and in program refinement calculi (Dijkstra [1975], Morris [1987], Morgan [1988], Back and von Wright [1990]).

Formal methods are used for code verification at the implementation level. Program verification was first developed by Floyd and Hoare (Floyd [1993], Hoare [1969]). Gaudel [1995] presents a theory of program testing based on formal specifications, which was developed into an important research topic. Hoare reports the use of formal assertions used at Microsoft for program testing rather than proving the correctness of programs (Hoare [2002]).

Formal methods are applied in software maintenance (Younger et al. [1996]) and software evolution (Ward and Bennett [1995]). The idea behind using formal methods during the design process is to improve the understanding of the system requirements and design (Craig et al. [1993]). This can be helpful by capture specification and design errors earlier in the design cycle. Early detection of errors leads to big time savings by reducing rework.

More information regarding successful application of formal methods in industry can be found in Clarke and Wing [1996]. Winter [2002] provides a successful example of how formal methods can be employed to enhance the industrial development process. Many standards in the field of railways safety currently mandate the use of formal methods in the design in order to verify correctness (Cimatti et al. [2012]). A review of formal methods relating to industrial projects can be found in Woodcock et al. [2009].

4.2 Process algebra

The aim of software verification is to assure that programs meet all the expected requirements. The way of providing a formal semantics of programming languages is a primary step toward program verification. In the last 40 years, many researcher have carried out research regarding this. The approach of process algebras provide a formal semantics for concurrent systems and allows to prove their properties (De Nicola [2014]).

The term “process algebra” was initiated in 1982 by Bergstra and Klop [1984b]. Process algebra is the way of using algebraic means to study distributed or parallel systems. The word “Process” here is used to represent the behaviour of a system. The word “Algebra” means that the approach for dealing with a system is algebraic and axiomatic (Baeten et al. [2007]). The main process algebra approaches are the Calculus of Communicating Systems (*CCS*) (Milner et al. [1992]), Communicating Sequential Processes (*CSP*) (Hoare [1983]) and Algebra of Communicating Processes (*ACP*) (Bergstra and Klop [1984a]).

4.2.1 CSP

The Process Algebra CSP (Abdallah [2005], Hoare [1983], Roscoe [1998], Ryan and Schneider [2001]) takes a prominent place among the formal method approaches, devoted to modelling, analysis and verification of concurrent systems. CSP has extensively used for modelling, verifying and the analysis of systems in industry.

McInnes [2007] developed a formal approach to engineering spacecraft behaviour, based on mathematical models using *CSP*. He successfully proves that process algebras can provide spacecraft designers with a mathematical approach for specifying and verifying system behaviour.

Winter [2002] studies railway interlocking systems by modelling the systems using CSP. The author uses the FDR2 checker to check the model against safety properties for example absence of derailment and collision. Winter successfully provides a case study of how the formal methods can support and develop the process in the industrial field.

Faber and Meyer [2006] prove that CSP-OZ-DC is suited for modelling systems in the industrial field. They assume critical system behaviours are specified by at least three aspects, namely the control flow aspect, the data aspect, and the real time aspect. CSP-OZ-DC combines CSP, Object-Z (OZ) and Duration Calculus(DC), where the internal and external behaviour is described by using CSP, the data aspects are represented using Object-Z, and the real time aspects are specified using the Duration Calculus. The

authors use this approach to model the emergency messages for the Radio Block Centres (RBCs) in the European Train Control System ETCS. RBCs manage the traffic in their area by exchanging the messages between the RBC and the trains in order to guarantee the safety of the railways in the ETCS. The author focuses on the emergency messages to ensure that trains never collide.

4.3 Theorem Provers

Theorem proving tools can be classified into one of two categories: interactive theorem proving or automatic theorem proving. An excellent introduction to the various flavours of theorem proving can be found in Harrison [2008], and a more technical analysis can be found in Boutin [1997]. In this thesis, our aim is to represent the process algebra CSP in the theorem prover Agda, in this stage we try to answer the question, why we use Agda, and not other theorem provers.

Software is becoming more important and popular in the modern community. It is also becoming more complicated. In our life, using computer programmes and safety-critical devices appearing in more and more every day. In these systems, the price of failure is rising as well. To efficiently develop programs, and be sure that they are correct, we should make the type system more expressive. Martin-Löf Type Theory (Martin-Löf [1975, 1982, 1984]) is a type theory which can be considered as a programming language equipped with an expressive type system.

Agda (Agda Community [2017a], Bove et al. [2009], Agda Community [2017b]) is a theorem prover and dependently typed programming language, which extends intensional Martin-Löf type theory (Martin-Löf [1984]). It is closely related to the theorem prover Coq (Coq Community [2015], Paulin-Mohring [2012], Coq Development Team [2015], Dowek et al. [1991]). Predicates are given as types, the elements of which are proofs of that property. The language of Agda is a functional programming language with numerous features added. Agda supports pattern matching, termination checking, inductive data types with inductive-recursive types, and inductive families. Induction-recursion allows to define data types and recursively defined functions simultaneously. In contrast, Coq supports inductive data types but not yet inductive-recursive types, which we use extensively in CSP-Agda for defining the set of choice sets and return types. Agda defines coinductive types in two approaches, where the newer one, which we use extensively, is based on coalgebras as given by their elimination rules, and the older one is based on coalgebras as given by introduction rules. Coq defines coinductive

type only in the way corresponding to the older way, namely by their introduction rules. Since we use extensively the definition of coinductive types by their elimination rules, Agda was the right choice to take in this thesis. Agda has a powerful mechanism for definitions by pattern and copattern matching, which is much more intuitive than Coq’s approach. Especially, when working with coalgebras, which are used a lot in CSP-Agda, the approach of Agda is much more intuitive. See chapter 3 for more information about this distinction.

Isabelle (Paulson [1994, 1988a]) is a generic proof assistant, originally developed at the University of Cambridge and the Technical University of Munich. This proof assistant allows mathematical formulas to be represented in a formal language, which allows the use of external tools linked to Isabelle to prove those formulas. Isabelle integrates the powerful automated theorem prover Sledgehammer (Paulson and Blanchette [2010]). Equality on coalgebras is defined as bisimilarity, which together with automated theorem proving support makes proving properties about coalgebras easy. Having all this support, Isabelle lacks dependent types. Since our approach was based on dependent types, our choice is Agda.

There are many functional languages supporting dependent types, for instance, McBride’s Epigram (McBride and McKinna [2004]), and Idris (Brady [2008, 2013, 2011]), developed by the group of Brady. Idris is primarily a programming language whereas we want to carry out proofs of properties of CSP-processes, therefore Idris is not suitable. Epigram is at the time of writing this thesis not as much supported as Agda, therefore our choice is Agda.

4.4 Defining Process Algebras in Functional Programming

There have been several successful approaches of combining functional programming with the process algebra CSP. Brown [2008] introduced a library (Communicating Haskell Process library, CHP) in Haskell. Since Haskell lacks explicit support for concurrency, he used a Haskell monad to provide a way to explicitly specify and control sequence and effects. In CHP the authors introduce a type (`CHP a`) of monadic processes with return value of type `a`. They have a return statement similar to our terminate process. In that paper they add operators from CSP such as (external) choice, parallelism, exception, sequencing and iteration. The focus is mainly on writing programs using these operators, not on creating a proper semantics and prov-



ing properties about their processes. Such a semantics is important to make sure that especially the terminate process is dealt with correctly – in our setting this gave rise to lots of subtle issues. Their setting doesn't seem to include the \checkmark -event, which plays an important role in CSP, and is quite difficult to deal with in a monadic setting, since one needs to add return values to \checkmark -events. It seems that they replace \checkmark transitions by τ -transitions to the terminated process. This doesn't work in CSP, since for instance in case of interleaving, \checkmark -transitions are blocked until both sides of a the interleaving operator have a \checkmark -transition, whereas τ -transitions can be followed by each process separately. We couldn't detect an explicit treatment of τ -transitions in their setting, although it is implicit in the internal choice operator. Brown [2009] present a new technique in order to generate CSP models of Haskell implementations using the CHP library. This approach is characterised by the need for a detailed semantics of the Haskell language. They use the FDR and ProB tools in order to check the model generated by this approach against deadlock, and as well to perform refinement checks. López et al. [2002] gave further examples of combining functional programming with process algebras. They used the functional program Eden in order to translate VPSPA specification into Eden programs. Eden extends Haskell, and can be considered as a concurrent functional language. Similarly, Fontaine [2011] gave another successful attempt of implementing the operational semantics of CSP (Roscoe [1998]) using the functional programming language Haskell. He presented a new tool for animation and model checking for CSP. Fontaine used a monad in order to model Input/Output, partial functions, state, non-determinism, monadic parser and passing of an environment. Tej and Wolff [1997] implemented the failures-divergence model of CSP developed by Brookes and Roscoe [1984] in Isabelle/HOL (Paulson [1986, 1988b]). They discovered an error which they corrected. Kammüller formalised CSP in Isabelle/HOL (Kammüller [2007]).

He used the features of the underlying higher order logic of Isabelle, for instance, the formalisation of fixed points due to Tarski and of data types. Kammüller used the predefined theory of the fixed point of Isabelle/HOL in order to define the semantics of recursive processes. In our approach, the predefined theory of coalgebras is used to define processes directly corecursively without having to use the recursion combinator. In contrast, Tej and Wolff [1997] recreated instead the entire Tarski fixed point theory.



4.5 Defining Process Algebras in Dependent Type Theory

Sellink [1994] gave a successful attempt to represent process algebras in type theory. Sellink used μCRL (Groote and Ponse [1995]), a language for reasoning about the Algebra of Communicating Processes, in order to implement it in the type theoretic proof assistant Coq (Paulin-Mohring [2012], Coq Development Team [2015]).

Sellink follows an algebraic approach towards proving laws about the process algebra ACP in Coq. Processes are defined from high level operators of ACP (which would correspond in CSP to forming processes from atomic ones using operations such as prefix, external choice, internal choice, interleaving etc). Then algebraic laws for these processes are axiomatised, from which one can derive new equalities. Translated into CSP it would mean to have laws regarding equalities and refinements of processes formed from these operations. Then one can show properties between processes formed using those operations. In this algebraic approach the processes are not directly implemented as something which could be executed, but kept abstract. In CSP-Agda we obtain processes one could program with, whereas in the approach by Sellink all we can do is prove properties about them. In our approach we can prove the algebraic laws, whereas in the approach by Sellink this cannot be done since these laws are the axioms of their approach.

Bezem et al. [1997] take the first step towards the formal verification of correctness proofs of real-life protocols in process algebras. The authors translated the proof theory of μCRL , which is based on the Algebra of Communicating Processes (ACP) of Bergstra and Klop, into Coq. As a case study, Bezem et al. verified the alternating bit protocol. Similarly, in Groote and van de Pol [1996], the process algebra μCRL is used to specify a protocol which describes the transmission of large data packets over channels. μCRL is a formal framework, which combines a process algebra and abstract data types. It was devised in 1990 to model and study the system. The correctness proof has been carried out in Coq. Cleaveland and Panangaden [1988] gave an earlier attempt to implement process algebras in type theory. They implemented the Calculus for Communicating Systems (Milner [1982]) in the proof assistant Nuprl (Aaron [2001]), which is similar to Agda, but is based on extensional Martin-Löf type theory. Elliott [2015] proposed an approach of representing concurrent programs in the dependently typed programming language Idris, and compiling them into the functional programming language Erlang using the Actor Model of Erlang. This allows to produce verified concurrent programs. Conceptually Erlang is similar to the

Occam programming language. CSP was highly influential in the design of Occam.

Close to our work, in the context of defining structures coinductively in dependent type theory, Spadotti [2015] described the implementation of a mechanised theory of regular trees coinductively in dependent type theory and implemented it in theorem prover Coq.

4.6 Defining Process Algebras using Coalgebras

Mathematical induction is a backbone of programming and program verification (Leino and Moskal [2014]). Briefly, induction is used to define algebraic data structures (Bird and Wadler [1988]), it arises behind program semantics, it is defined to obtain finite iteration and recursion (Beckert et al. [2007]), and used in order to carry out proofs (Sheeran et al. [2000]). Induction deals with finite, or more generally well-founded behaviour, whereas, in contrast, coinduction deals with an infinite objects (infinite data, infinite behaviour, etc.). Coinduction is the dual of induction. Coinduction is essential in order to verify and program infinite programs, which might go on forever. For instance, coinduction can be used to construct data types (Jones [2003]), define semantics (Park [1981]) and to carry out proofs (Leroy [2009]). Indubitably, both induction and coinduction are desirable to developing programs and verifying them.

Coalgebras have become one of the main methods for specifying the reactive behaviour of a system. A good overview on coalgebras can be found in Rutten [2000]. Goncharov and Schröder [2013] defined a framework for concurrent processes, where atomic steps have side effects. Goncharov et al. used the monadic principle in order to encapsulate effects. Processes in that approach are modelled as infinite resumptions using a final coalgebra. The main result of this paper is a corecursion scheme over the base language and a new semantics for operators on processes such as parallel composition. They extended the framework to cover safety properties.

Mossakowski et al. [2006] gave a good example of using coalgebras in order to extend the specification language CASL. Goncharov et al. [2014] also developed a semantic framework that combines monads, operations and recursive definitions. Their metalanguage for effectful recursion definitions was inspired by Moggi's computational metalanguage. They integrated the coalgebraic and monad aspects of the computations into a single framework. Using the notion of a complete Elgot monad, the authors developed a met-

alanguage. The work closest to our research is Goncharov et al. [2014], who have also formalised corecursive definitions of process algebraic operations on processes with side effects using a new metalanguage.

Gimenez [1995] represent a calculus for describing broadcasting systems (Prasad [1995]) in the theorem prover Coq, such that the behaviour of processes is modelled as coinductive types.

Isobe and Roggenbach [2005] have developed a tool called CSP-Prover, which allows to carry out refinement proofs in CSP, specifically proofs for infinite state systems. CSP-prover is an interactive theorem prover, which is built upon the theorem prover Isabelle/HOL. They implemented the theories of complete metric spaces (cms) and complete partial orders (cpo) in Isabelle/HOL in order to model infinite state systems in CSP-prover. In CSP-Agda, in contrast, the semantics of processes is defined as a coinductively defined predicate rather than a set, which allows to reason directly using the definition of those predicates.

4.7 Agda as Platform for Modelling Programs

Kanso in his PhD thesis Kanso [2012] developed a framework for the development of verified railway interlocking in the theorem prover Agda. In this thesis, Kanso's integrated automated theorem proving (SAT solving and CTL model-checking) into type theory. He addressed verification in the railway domain and was applying it to both modern electronic interlockings and traditional mechanical interlockings. Kanso formalises railway interlocking systems in ladder logic, namely as a large set of Boolean variables together with a next step operation, which defines the state of the Boolean variables in the next step. For instance we could have a variable corresponding to a request to set a signal to green, where the state of the signal is represented by another (or several) Boolean variables. The next step function would then set, in case the request was made and it is safe to set a signal to green, actually set that variable so that the signal is green. In the CSP approach, the units in a railway interlocking system are defined instead as processes, which communicate with each other. In Kanso's approach these processes would be translated into states of Boolean variables, and the next communications possible in the process world would correspond to computing the variables representing the next state from the current one. In the ladder logic approach everything is deterministic. Non determinism can only be represented by having input variables which decide which of the non-deterministic choices to carry out.

Chapter 5

The Library CSP-Agda

In this chapter we will first introduce the way of defining label where processes depend on it (Sect. 5.1). Then we show how to add a monad structure to CSP (Sect. 5.2). Then (Sect. 5.3) we show how to represent CSP processes in Agda in a coalgebraic way. Then we develop the representation of CSP operators in CSP-Agda: sequential composition or monadic bind (Sect. 5.5), the recursion operator (Sect. 5.6), STOP, SKIP, Terminate, and DIV (Sect. 5.7), prefix (Sect. 5.8), internal choice (Sect. 5.9), external choice (Sect. 5.10), renaming (Sect. 5.11), interleaving (Sect. 5.13), hiding (Sect. 5.12), parallel operator (Sect. 5.14), and, finally, the interrupt operator (Sect. 5.15).

5.1 Label Universe LUniv

Processes depend on a set of labels. We define a type `LUniv` of labels together with an equality, proofs of reflexivity, symmetry, the transfer principle, a show function, and a list of labels, intended to be the list of all labels available (see file `labelUniv.agda`, Appendix A.51). The transfer principle expresses that if we have any predicate on labels, two labels l and l' and a proof that they are equal, then we can transfer an element of $Q\ l$ to $Q\ l'$. Since the set of labels will be in our setting finite, one could have used as well the intensional equality together with a proof that it is decidable. Decidability of equality is needed, since when simulating a CSP program, one needs to decide which process to continue with after an external choice has been provided to the process. The component `sym==lf` can actually be derived from `transf` and `refl==l`. `⊤`, which was introduced in SubSect. 3.2.6, was renamed to `⊤'`, in order to avoid conflicts with imported libraries:

```
record LUniv : Set1 where
  field
    Labelf    : Set
```

```

_==lf_    : Labelf → Labelf → Bool
refl==lf  : {l : Labelf} → T' (l ==lf l)
sym==lf   : {l l' : Labelf} → T' (l ==lf l') → T' (l' ==lf l)
transf    : {l l' : Labelf} → (Q : Labelf → Set)
           → T' (l ==lf l') → Q l → Q l'
showLabelf : Labelf → String
LabelListf : List Labelf

```

For type inference purposes it is better to introduce a separate type of labels for an element of `LUniv` – otherwise Agda is not able to guess which element of `LUniv` is chosen:

```

data Label (lu : LUniv) : Set where
  lab : LUniv.Labelf lu → Label lu

```

We now transfer the operations from `LUniv` to `Label`:

```

_==l_ : {lu : LUniv} → Label lu → Label lu → Bool
_==l_ {lu} (lab x) (lab y) = LUniv._==lf_ lu x y

refl==l : {lu : LUniv} {l : Label lu} → T' (l ==l l)
refl==l {lu} {lab x} = LUniv.refl==lf lu {x}

sym==l : {lu : LUniv} {l l' : Label lu} → T' (l ==l l') → T' (l' ==l l)
sym==l {lu} {lab x} {lab y} p = LUniv.sym==lf lu {x} {y} p

transfLu : {lu : LUniv} (Q : Label lu → Set) {l l' : Label lu}
  → T' (l ==l l') → Q l → Q l'
transfLu {lu} Q {lab l} {lab l'} ll' q =
  LUniv.transf lu {l} {l'} (λ x → Q (lab x)) ll' q

_==L_ : {lu : LUniv} → Label lu → Label lu → Set
_==L_ {lu} l l' = T' (_==l_ {lu} l l')

showLabel : {lu : LUniv} → Label lu → String
showLabel {lu} (lab x) = LUniv.showLabelf lu x

LabelList : (lu : LUniv) → List (Label lu)
LabelList lu = map (λ x → lab x) (LUniv.LabelListf lu)

```

```

labelBoolFunToString : {lu : LUniv} → (Label lu → Bool) → String
labelBoolFunToString {lu} f = unlines (map (showLabel {lu})
                                           (filter f (LabelList lu)))

labelLabelFunToString : {lu : LUniv} → (Label lu → Label lu) → String
labelLabelFunToString {lu} f = "[["
    ++s unlinesWithChosenString ", "
      (map (λ l → showLabel {lu} (f l))
           (LabelList lu))
    ++s " <- " ++s showLabel {lu} l
    ++s "]]"

```

As an example we define a set of labels having three labels (defined in file `label.agda`, Appendix A.49).

```

data LabelSimple : Set where
  laba labb labc : LabelSimple

```

```

trl : {l l' : LabelSimple} → (Q : LabelSimple → Set)
    → T' (l ==lsimpl l') → Q l → Q l'
trl {laba} {laba} Q t q = q
trl {laba} {labb} Q () q
trl {laba} {labc} Q () q
trl {labb} {laba} Q () q
trl {labb} {labb} Q t q = q
trl {labb} {labc} Q () q
trl {labc} {laba} Q () q
trl {labc} {labb} Q () q
trl {labc} {labc} Q t q = q

symLabelSimple : {l l' : LabelSimple} → T' (l ==lsimpl l')
    → T' (l' ==lsimpl l)
symLabelSimple {laba} {laba} tt = tt
symLabelSimple {laba} {labb} ()
symLabelSimple {laba} {labc} ()
symLabelSimple {labb} {laba} ()
symLabelSimple {labb} {labb} tt = tt

```

```

symLabelSimple {labb} {labc} ()
symLabelSimple {labc} {laba} ()
symLabelSimple {labc} {labb} ()
symLabelSimple {labc} {labc} tt = tt

lsimple : LUniv
Labelf lsimple = LabelSimple
_==lf_ lsimple = _==lsimpl_
refl==lf lsimple {l} = refl==lsimple {l}
showLabelf lsimple = showLabelSimple
LabelListf lsimple = LabelListSimple
transf lsimple = trl
sym==lf lsimple {l} {l'} = symLabelSimple {l} {l'}

```

5.2 Monadic Composition in CSP-Agda

In standard process algebras, if a process terminates, it does not return any information except for that it terminated.¹ We want to define processes in a monadic way in order to combine them in a modular way. Therefore, if processes terminate, they should return some additional information, namely the result returned by the process.

In functional programming, a monad is given by a functor M together with morphisms $\gg= : M A \rightarrow (A \rightarrow M B) \rightarrow M B$ and $\text{return} : A \rightarrow M A$ such that the following laws hold:

$$\begin{aligned}
\text{return } a \gg= f &= f a \\
p \gg= \text{return} &= p \\
(p \gg= f) \gg= g &= p \gg= (\lambda x. f x \gg= g)
\end{aligned}$$

The type of interactive programs can be considered as a monad in the following way:

- For a given set A , $(M A)$ is the set of interactive programs, which may or may not terminate, and if they terminate, they will return a result $a : A$.
- Assume P is program in $(M A)$, and Q is a function, which for $a : A$ returns a program in $(M B)$. Then $P \gg= Q$ is the program, which

¹See below for a discussion on terminated processes vs terminating events as they occur in CSP.

executes as follows: First P is executed. If P terminates with result a then we continue executing $(Q\ a)$. The result of the whole process is the result of $(Q\ a)$ (if it terminates).

- The program `(return a)` will terminate without any interaction with result a .

Processes in our approach are similar to interactive programs. They are defined using an atomic operation, which corresponds to a one step interaction in interactive programs, and describes the next transition a process can make. We have the terminated process `(terminate a)` which plays the rôle of `(return a)` in interactive processes. We can monadically combine processes in a similar way as what we can do with interactive programs. Since processes can loop for ever, they are defined coinductively – again the same occurs when representing interactive programs in Agda. The standard CSP-operators are in our approach defined rather than atomic as in process algebras. Since processes are given coinductively, we can introduce processes by primitive corecursion (also called guarded recursion). The principle of primitive corecursion, which is enforced by Agda’s termination checker, will guarantee processes to be productive, which means for a process we can determine, whether it has terminated or not, and, in case it has terminated, the result returned, and, in case it hasn’t terminated, which next transitions it can make, and the next processes after firing these transitions.

We note here already that laws of CSP will usually not hold with respect to definitional or intensional equality. We will introduce various propositional equalities on CSP processes later, and prove algebraic laws for CSP processes by referring to those semantic equalities.

Terminated processes vs termination events. In CSP termination is handled by events. A process can terminate, which is modelled by an event with reserved label \checkmark . If $P \xrightarrow{\checkmark} P'$, then P' is a deadlocked process, which is in all standard semantic models of CSP equal to the process `STOP`. A first step towards a monadic version of processes is that we add a return value to \checkmark -transitions. This is the result returned when the process terminates, which can be used for choosing a continuation e.g. in monadic `bind`. Since P' is equal to `STOP` we can omit it and just write $P \xrightarrow{\checkmark, a}$ for P having a termination event with return value a .

Adapted to the monadic setting, we have the CSP process `(SKIP a)` (for a return value a), which has as only transition `SKIP $a \xrightarrow{\checkmark, a}$` . We want to have as well a terminated process `(terminate a)` with result a , which is our name for the monadic `(return a)`. `(terminate a)` is very similar to `(SKIP a)`, except

that `(terminate a)` has terminated, whereas `(SKIP a)` will terminate.

If we lift (\circ) to a monadic $\gg=$ we get `SKIP a` $\gg= Q \xrightarrow{\tau} Q a$, whereas we want definitionally `terminate a` $\gg= Q = Q a$ without a τ -transition. Semantically this doesn't make a difference, since in the various semantics of CSP we have $\tau \longrightarrow P = P$. When using it, it makes a difference, since when composing processes we don't want a τ -transition in between.

Because of the equation $\tau \longrightarrow P = P$, we could use `(SKIP a)` for `(terminate a)` and optimise the rules to guarantee `SKIP a` $\gg= Q = Q a$. However, this makes the code very complex. Experience shows, that a lot of care needs to be taken in Agda when writing programs, because otherwise verification becomes very complex, since it needs to deal with any special cases introduced when defining the programs. Therefore, it seems to be a better approach to have a separate process `(terminate a)`. That process will be in CSP semantics equal to `(SKIP a)`. When defining CSP operators applied to arguments, we define it for the argument `(terminate a)` in the same way as for the argument `(SKIP a)`. However, if the result is a process, which has only one τ -transition to a process P , we return instead, when the argument is `(terminate a)`, directly process P without the τ -transition. So we have two kinds of terminated processes in CSP-Agda. One is the result of following a termination event \checkmark , and one is the new terminated process `(terminate a)`.

A process which terminates behaves as well differently from `(terminate a)`: in CSP, if $P \xrightarrow{\checkmark} P'$ then $P \circ Q \xrightarrow{\tau} Q$, i.e. a τ -transition is needed before passing on control to Q . This should be reflected as well in CSP-Agda, whereas we want that the equation `(terminate a)` $\gg= Q = Q a$ holds without any τ -transition in between. Semantically, this doesn't make a difference, since in CSP we have $\tau \longrightarrow P = P$. However, when actually running processes we don't want to have this τ -transition.

This is needed, since when modelling recursion or composition we want to be able to continue without a τ -transition with the next process. One could argue that in CSP-semantics $\tau \longrightarrow P = P$, but when composing processes in a modular way you do not want to introduce artificial τ -transitions.

One could say that `(terminate a)` is process, which immediately terminates, and passes immediately control to any other processes. For composing processes in a modular way one uses `(terminate a)`.

When modelling CSP-operators, we will model the operational rules of CSP. We will treat `terminate` similar to a process, which issues a termination event, except that we force this termination event to be executed – so other processes cannot execute an external or internal choice operation before we have dealt with this \checkmark -transition. Furthermore we pass immediately control to other processes without requiring a τ -transition. This seems to be the

right approach to guarantee monadic compositionality.

Finally, we note that in CSP a process can allow both \checkmark -transitions and external choices and internal choices, and we will model this in CSP-Agda as well.

5.3 Representing CSP Processes in Agda

In a monadic version, a process $P : \text{Process } A$ is either a terminating process (`terminate a`), which has return value $a : A$, or it is process (`node P`), which progresses. Here $P : \text{Process+ } A$, where $(\text{Process+ } A)$ is the type of progressing processes. A progressing process can proceed at any time with labelled transitions (external choices), silent transitions (internal choices), or \checkmark -events (termination). After a \checkmark -event, the process becomes deadlocked, so there is no need to determine the process after that event. However, as discussed before we will add a return value $a : A$ to \checkmark -events.

Elements of $(\text{Process+ } A)$ are therefore determined by

- (1) an index set `E` of external choices and for each external choice e the Label $(\text{Lab } e)$ and the next process $(\text{PE } e)$;
- (2) an index set of internal choices `I` and for each internal choice i the next process $(\text{PI } i)$; and
- (3) an index set of termination choices `T` corresponding to \checkmark -events and for each termination choice t the return value $\text{PT } t : A$.

One might consider reducing the number of components by unifying the choice sets and adding τ and \checkmark to the set of labels. However, the operators of CSP handle external, internal, and termination transitions quite differently. If we encoded them as one choice set, we would for each operator have to select the choices corresponding to these categories, form the new choices and recombine them. Keeping them as separate entities makes programming and then verification much easier.

We define $(\text{Process+ } A)$ as a record. Definition of elements of it by copattern matching is very convenient, since it avoids the need to define the components of $(\text{Process+ } A)$ as auxiliary functions as one would have to do when using `data`.

Processes need to be defined coinductively instead of inductively – otherwise processes would always after finitely many transitions eventually terminate. Processes will therefore be defined by primitive corecursion or guarded recursion. The left hand side of a primitive corecursion scheme needs to have an observation applied to the element of the coinductive type to be

defined. We could do this using $(\text{Process}+ A)$. However, this would mean that for defining a process by primitive corecursion, we need to define all the 7 components. It seems to be natural to define processes by primitive corecursion where we want to equate the result of applying an eliminator to a process directly with a process formed from other processes, without having to define all 7 components. Because of this we introduce a third type of processes, $(\text{Process}\infty A)$, which has a observation forcep returning an element of $(\text{Process} A)$.

We will develop a simulator for processes, which displays the evolving of processes following external and internal choices. The simulator needs to display processes as strings. Since processes are infinite objects, we cannot directly compute such finite strings. The solution is to add a new field $\text{Str}+$ to $(\text{Process}+ A)$, which determines the string. That string needs to be user-defined. We need to add as well a field $\text{Str}\infty$ to $(\text{Process}\infty A)$. The reason is that we can only use $\text{Str}+$ to obtain a string from an element of $(\text{Process}\infty A)$, if we have a smaller size available, which in general is not the case. This is no artificial restriction imposed by sizes: without this field it is in general not possible to compute a string. For instance, we could define elements of $(\text{Process}\infty A)$ corecursively without assigning to it a string directly. Then any string computed would need to be infinite.

We model the sets of external, internal, and termination choices as elements of an inductive-recursively defined universe Choice . Elements c of Choice are codes for finite sets, and $(\text{ChoiceSet } c)$ is the set it denotes. In addition we define a string $(\text{choice2Str } c)$ representing c , and a function choice2Enum , which computes from c a list of all choices. This will be used to print a list of choices in the simulator for CSP processes.

We require as well that the set of return values are elements of Choice . This allows us to print the result returned when a process terminates. However, for the return types it is not needed that they are finite sets. We plan to introduce in future a separate universe for return values, where we only require that a string can be computed for each element, but drop the requirement to compute an enumeration of its elements. Then we could have the set of natural numbers as a return value, which could be useful for defining processes by recursion over the natural numbers.

The resulting code for processes in Agda is as follows:

Definition 5.3.1 (Agda Definition)

mutual

record $\text{Process}\infty (i : \text{Size}) \{lu : \text{LUniv}\}(c : \text{Choice}) : \text{Set}$ **where**
coinductive

```

field
  forcep : {j : Size< i} → Process j {lu} c
  Str∞ : String

data Process (i : Size) {lu : LUniv} (c : Choice) : Set where
  terminate : ChoiceSet c → Process i {lu} c
  node      : Process+ i {lu} c → Process i {lu} c

record Process+ (i : Size) {lu : LUniv} (c : Choice) : Set where
  constructor process+
  coinductive
  field
    E   : Choice
    Lab : ChoiceSet E → Label lu
    PE  : ChoiceSet E → Process∞ i {lu} c
    I   : Choice
    PI  : ChoiceSet I → Process∞ i {lu} c
    T   : Choice
    PT  : ChoiceSet T → ChoiceSet c
    Str+ : String

delay : {i : Size} → {lu : LUniv} → {c : Choice} → Process i {lu} c
      → Process∞ (↑ i) {lu} c
forcep (delay P) = P
Str∞ (delay P) = Str P

```

An example of a process is as follows:

```

P = node (process+ E Lab PE I PI T PT "P")
    : Process String where
E = code for {1,2}      I = code for {3,4}
T = code for {5}
Lab 1 = a      Lab 2 = b      PE 1 = P1
PE 2 = P2    PI 3 = P3    PI 4 = P4
PT 5 = "STOP"

```

The universe of choices is given by a set **Choice** of codes for choice sets, and a function **ChoiceSet**, which maps a code to the choice set it denotes. Universes were introduced by Martin-Löf (e.g. Martin-Löf [1984]) in order to formulate the notion of a type consisting of types. Universes are defined in

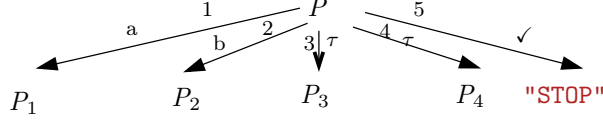


Figure 5.1: Process In CSP-Agda

Agda by an inductive-recursive definition (Dybjer [1992, 1991, 2000], Dybjer and Setzer [2003]): we define inductively the set of codes in the universe while recursively defining the decoding function.

We give here the code expressing that Choice is closed under `fin`, `⊕`, `×`, `subset`, `Σ` and `namedElements`. Closure under other operations can easily be added as needed. The type `(NamedElements l)` is essentially `(Fin (length l))`. The function `choice2Str` will for elements of this set print the n th element of l , giving them more meaningful names. We don't equate `(NamedElements l)` with `(Fin (length l))`. This facilitates type inference.

The type `subset A f` is the set of $a : A$ such that $(f a)$ is true. It is defined in Agda as follow:

```
data subset (A : Set) (f : A → Bool) : Set where
  sub : (a : A) → T (f a) → subset A f
```

We could have defined `Choice` simply as the collection of finite sets `(Fin n)`. However, then the indices of choice sets would lose connection with the actual types constructed. For instance in case of external choice $P \sqcup Q$, in our setting a choice `(inj1 x)` refers to P , and a choice `(inj2 x)` refers to Q .

```
data NamedElements (s : List String) : Set where
  ne : Fin (length s) → NamedElements s
```

```
mutual
```

```
data Choice : Set where
  fin : ℕ → Choice
  _⊕_ : Choice → Choice → Choice
  _×_ : Choice → Choice → Choice
  subset' : (E : Choice) → (ChoiceSet E → Bool) → Choice
  Σ' : (E : Choice) → (ChoiceSet E → Choice) → Choice
  namedElements : List String → Choice
  list : (E : Choice) (l : List (ChoiceSet E)) → Choice
```

```
ChoiceSet : Choice → Set
```

```

ChoiceSet (fin n) = Fin n
ChoiceSet (s ⊔' t) = ChoiceSet s ⊔ ChoiceSet t
ChoiceSet (E ×' F) = ChoiceSet E × ChoiceSet F
ChoiceSet (Σ' A B) = Σ[ x ∈ ChoiceSet A ] ChoiceSet (B x)
ChoiceSet (namedElements s) = NamedElements s
ChoiceSet (subset' E f) = subset (ChoiceSet E) f
ChoiceSet (list E l) = ListChoiceElements E l

```

```

data ListChoiceElements (E : Choice)(l : List (ChoiceSet E)) : Set where
  lce : Fin (length l) → ListChoiceElements E l

```

The constructor Σ' is not used later in the code for this thesis. It could be used if we wanted to have generalised operators for e.g. external internal choice

$$\Box_{x : \text{ChoiceSet } c} P_x$$

Then the external choice set would be

$$\Sigma_{x : \text{ChoiceSet } c} (E' x)$$

which in Agda would be written as

$$\Sigma[x \in \text{ChoiceSet } c](E' x)$$

Here $(E' x)$ is the set of external choices for $(P x)$.

An alternative approach could be to use `NamedElements` which allows introducing customary names for each element. But that might be computationally expensive because it would for instance in case of the product require to map elements of $\text{Fin } (n \times m)$ to $\text{Fin } (n) \times \text{Fin } (m)$. Therefore we believe that our approach is a good choice.

Some remarks on our design decisions: In original CSP termination is modelled by a transition labelled with the special termination event \checkmark , after which no longer any transitions are possible. This means that a process might have both labelled/silent transitions, and can terminate at the same time. The process `(terminate a)` is a process, which has terminated, and having the possibility to terminate will be modelled in our setting by a τ -transition to the process `(terminate a)`.

We allow both the internal choice set and external choice set to be empty, in this case the process deadlocks. Note that this is a process different from the process `(terminate a)`, which terminates explicitly.

`(Lab P)` can return the same value for different elements of `(Lab P)`, therefore a process can have several transitions with the same label. This

is in accordance with CSP. One could instead demand that for each label there is at most one transition possible, and replace processes having several transitions with the same label by one, which has one transition followed by silent transitions to the different choices.

5.4 Three Kinds of Termination of Processes

There are three kinds of versions of termination in CSP-Agda:

- The terminated process (**terminate** a), which has terminated. When monadically composed with another process Q , it will definitionally reduce to $(Q\ a)$.
- The process which can make a τ -transition to (**terminate** a). This process can choose to make a transition. This process is actually in all semantics equal to (**terminate** a). Since $\tau \xrightarrow{P}$ is equal to P . However, processes can have τ -transitions to (**terminate** a) and as well other internal, external choices or termination events.
- The process which can make a termination event with value a . This process will according to the rules of CPS behave different from a process with τ -transition to (**terminate** a): When combined using the interleaving with another process, the termination event can only be executed in sync with a termination event of the other processes, whereas τ -transitions can be executed independently of each other.

We note here that it is possible for a process to have both internal choices and termination event. An example would be $SKIP \sqcap P$ where $P \xrightarrow{\tau} SKIP$ (e.g. P is the result of hiding a from $a \longrightarrow SKIP$). It has a τ -transition to $SKIP \sqcap SKIP$ and a \checkmark -event.

5.5 Sequential Composition

In CSP the semi-colon operator (\circledast) is used for sequencing two processes, where, if the first process terminates, control is passed to a second one. The rules for sequential composition in CSP are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P}}{P \circledast Q \xrightarrow{\tau} Q} \quad \frac{P \xrightarrow{\mu} \bar{P}}{P \circledast Q \xrightarrow{\mu} \bar{P} \circledast Q} [\mu \neq \checkmark]$$

In CSP-Agda we have monadic composition $P \gg= Q$, where Q depends on the return value of P . The monadic bind $(P \gg= Q)$ allows to compose two processes P and Q while allowing the second process depend on the return type c_0 of P . So Q has an extra argument of the return type (`ChoiceSet` c_0).

Let us consider first the version $_ \gg= + _$ where the first process is an element of set of progressing processes `Process+`. The transitions of $(P \gg= + Q)$ are as follows: It follows first external and internal choices of P . If P is the terminated process with return type a , the process continues as process $(Q\ a)$. A special case is a termination event in P with return value a . Following the operational semantics of CSP, $(P \gg= + Q)$ has in this case an internal choice (i.e. a τ -transition) to process $(Q\ a)$.

In total, $(P \gg= + Q)$ has two possible internal choice events, namely internal choices of P and termination events of P . It has no termination events.

More precisely, in $P' := (P \gg= + Q)$, external choices of P become external choices of P' using a recursive call, similarly for internal choices. For termination events of P with return value `PT` $P\ c = a$, we get additional internal choice transitions $P' \xrightarrow{\tau} Q\ a$.

In case of the monadic bind $_ \gg= _$ on `Process`, we have a special case, when $P = \text{terminate}\ x$. In this case $P \gg= Q$ is equal to $(Q\ x)$ (one needs to apply `forcep` in order to obtain an element of `Process`). This is different from termination events for P , where a silent transition is required before obtaining $(Q\ x)$.

We obtain therefore in monadic form `SKIP` $a \gg= Q \xrightarrow{\tau} Q\ a$ as only transition. $(\text{terminate}\ a)$ should behave as `(SKIP` $a)$, however we omit unnecessary τ -transitions. Therefore we define `terminate` $a \gg= Q = Q\ a$. If P has a \checkmark , a -event, we cannot define $P \gg= Q = Q\ a$, since P could have other external or internal choices. Therefore, a τ -transition before continuing with $(Q\ a)$ will be added, as it happens in original CSP.

In case of progressing processes $P \gg= Q$ makes a direct call to $_ \gg= + _$. The function $_ \gg= \infty _$ makes as well a direct call to $_ \gg= + _$.

The full definition of monadic bind is as follows:

```
\_ \gg= Str \_ : {c0 : Choice} -> String
              -> (ChoiceSet c0 -> String) -> String
s \gg= Str f = s ++ s ">>" ++ s choice2Str2Str f
```

mutual

```
\_ \gg= \infty \_ : {i : Size} -> {lu : LUniv} -> {c0 c1 : Choice}
              -> Process\infty i {lu} c0
              -> (ChoiceSet c0 -> Process\infty i {lu} c1)
```

$$\begin{aligned}
& \rightarrow \text{Process}_\infty i \{lu\} c_1 \\
\text{forcep } (P \gg=\infty Q) &= \text{forcep } P \gg= Q \\
\text{Str}_\infty (P \gg=\infty Q) &= \text{Str}_\infty P \gg=\text{Str} (\text{Str}_\infty \circ Q) \\
\\
_ \gg= _ : \{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\
& \rightarrow \text{Process } i \ c_0 \\
& \rightarrow (\text{ChoiceSet } c_0 \rightarrow \text{Process}_\infty (\uparrow i) \{lu\} c_1) \\
& \rightarrow \text{Process } i \ c_1 \\
\text{node } P \gg= Q &= \text{node } (P \gg=+ Q) \\
\text{terminate } x \gg= Q &= \text{forcep } (Q \ x) \\
\\
_ \gg=+ _ : \{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\
& \rightarrow \text{Process}+ i \ c_0 \\
& \rightarrow (\text{ChoiceSet } c_0 \rightarrow \text{Process}_\infty i \{lu\} c_1) \\
& \rightarrow \text{Process}+ i \ c_1 \\
\text{E } (P \gg=+ Q) &= \text{E } P \\
\text{Lab } (P \gg=+ Q) &= \text{Lab } P \\
\text{PE } (P \gg=+ Q) \ c &= \text{PE } P \ c \gg=\infty Q \\
\text{I } (P \gg=+ Q) &= \text{I } P \ \Psi' \ \text{T } P \\
\text{PI } (P \gg=+ Q) (\text{inj}_1 \ c) &= \text{PI } P \ c \gg=\infty Q \\
\text{PI } (P \gg=+ Q) (\text{inj}_2 \ c) &= Q (\text{PT } P \ c) \\
\text{T } (P \gg=+ Q) &= \emptyset' \\
\text{PT } (P \gg=+ Q) () & \\
\text{Str}+ (P \gg=+ Q) &= \text{Str}+ P \gg=\text{Str} (\text{Str}_\infty \circ Q)
\end{aligned}$$

In the above code `choice2Str2Str` converts a function $\text{ChoiceSet } c \rightarrow \text{String}$ into a meaningful string, making a case distinction on the argument. Furthermore, \emptyset' is an abbreviation for `fin 0`.

We note here that sequential composition doesn't require the use of sized types. However, it is important to know that it is size preserving. This allows to apply it in other corecursive definitions to the coinduction hypothesis which we will do frequently.

5.6 The Recursion Operator

We can define recursion in a similar way to $_ \gg= _$. The operation takes an $s : \text{String}$, $f : \text{ChoiceSet } c_0 \rightarrow \text{Process}+ i (c_0 \ \Psi' \ c_1)$ and an $a : \text{ChoiceSet } c_0$ and returns a process (`rec s f a`), which operates as follows: We start with process $(f \ a)$ and follow its external and internal choices. If it terminates with result $(\text{inj}_2 \ x)$, the recursion terminates with result x . If it terminates

with result ($\text{inj}_1 a'$), we recursively start again, with a replaced by a' .

However, in case $(f x)$ terminates immediately, this procedure (unless we put a τ -transition after each loop iteration) will result potentially in a black hole recursion. To avoid this we require $f x : \text{Process+}$, which is the type of processes, which have not terminated. Because of this, the result of `rec` is productive. We have an argument s , which is the name of the resulting process, since an automatically generated name would in most cases be unreadable.

The Agda code is as follows (`(renameP name P)` renames the `Str+` component of process P to $name$):

```
mutual
  rec : {i : Size} → {lu : LUniv}{c0 c1 : Choice}
    → (s : String)
    → (ChoiceSet c0 → Process+ (↑ i) {lu} (c0 ⊎' c1))
    → ChoiceSet c0
    → Process∞ i {lu} c1
  forcep (rec s f a) = renameP s
    (f a >>=+p recaux s f)
  Str∞ (rec s f a) = s

  recaux : {i : Size} → {lu : LUniv}{c0 c1 : Choice}
    → (s : String)
    → (ChoiceSet c0 → Process+ (↑ i) {lu} (c0 ⊎' c1))
    → (ChoiceSet c0 ⊎ ChoiceSet c1)
    → Process∞ i {lu} c1
  recaux s f (inj1 x) = rec s f x
  recaux s f (inj2 x) = delay (terminate x)

  recStr : {lu : LUniv}{c0 : Choice}
    → (ChoiceSet c0 → String)
    → ChoiceSet c0 → String
  recStr f a = "rec(" ++s choice2Str2Str f ++s ", " ++s choice2Str a ++s ")"
```

Here `delay` lifts an element of `Process` to `Process∞`:

```
delay : {i : Size} → {lu : LUniv} → {c : Choice} → Process i {lu} c
  → Process∞ (↑ i) {lu} c
forcep (delay P) = P
Str∞ (delay P) = Str P
```



5.7 STOP, SKIP, Terminate, DIV

The STOP process in CSP is the deadlocked process, which refuses all communication. It has no transition rule. It can be modelled as a process, which has empty external, internal and termination choice sets \emptyset' . The components Lab, PE, PI, PT have as domain the empty set, and can be given by the function `efq` (for ex falso quodlibet) which is defined by the empty case distinction, as denoted by `()`. The name of STOP is "STOP".

`efq` : $\{A : \text{Set}\} \rightarrow \text{Fin } 0 \rightarrow A$
`efq` ()

`STOP+` : $\{i : \text{Size}\} \rightarrow (c : \text{Choice}) \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process+ } i \{lu\} c$
`STOP+ c = process+ \emptyset' efq efq \emptyset' efq \emptyset' efq "STOP"`

`STOP` : $\{i : \text{Size}\} \rightarrow (c : \text{Choice}) \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process } i \{lu\} c$
`STOP c = node (STOP+ c)`

`STOP ∞` : $\{i : \text{Size}\} \rightarrow (c : \text{Choice}) \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process}\infty i \{lu\} c$
`forcep (STOP ∞ c) = STOP c`
`Str ∞ (STOP ∞ c) = "STOP ∞ "`

The CSP process SKIP terminates immediately. Its only transition is

$$\text{SKIP} \xrightarrow{\checkmark} \text{STOP}$$

In CSP we have that $\text{SKIP} \circ P \xrightarrow{\tau} P$ instead of $\text{SKIP} \circ P = P$. Therefore SKIP is not the process (`terminate a`) but a process, which has no external or internal choices and only one \checkmark choice with a given return value. Let $\top' = \text{fin } 1$ be the one element choice set. SKIP is defined as follows:

`SKIP+` : $\{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c)$
 $\rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process+ } i \{lu\} c$
`SKIP+ a = process+ \emptyset' efq efq \emptyset' efq \top' ($\lambda _ \rightarrow a$)`
`("SKIP(" ++s choice2Str a ++s ")")`

`SKIP` : $\{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c)$
 $\rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process } i \{lu\} c$



```
SKIP a = node (SKIP+ a)
```

```
SKIP $\infty$  : {i : Size}  $\rightarrow$  {c : Choice}  $\rightarrow$  (a : ChoiceSet c)
            $\rightarrow$  {lu : LUniv}  $\rightarrow$  Process $\infty$  i {lu} c
forcep (SKIP $\infty$  a) = SKIP a
Str $\infty$  (SKIP $\infty$  a) = ("SKIP $\infty$ (" ++s choice2Str a ++s ")")
```

We have as well the terminating process given by

```
terminate: {i : Size}  $\rightarrow$  (c : Choice)  $\rightarrow$  (a : ChoiceSet c)
            $\rightarrow$  Process+ i c
```

Direct divergence in the sense of black hole recursion does not occur in CSP-Agda, since productivity is guaranteed by Agda's termination checker. Note that in case of recursion, productivity is guaranteed by referring to the type of not-terminated processes `Process+`. However one can easily define a process, which has infinitely many τ transitions to itself:

mutual

```
DIV $\infty$  : {i : Size}{lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  Process $\infty$  i {lu} c
forcep DIV $\infty$  = DIV
Str $\infty$  DIV $\infty$  = "DIV"
```

```
DIV : {i : Size}{lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  Process i {lu} c
DIV = node DIV+
```

```
DIV+ : {i : Size}{lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  Process+ i {lu} c
DIV+ = (process+  $\emptyset'$  efq efq  $\top'$  ( $\lambda$  _  $\rightarrow$  DIV $\infty$ )  $\emptyset'$  efq "DIV")
```

5.8 Prefix

The prefix operator $a \longrightarrow P$ has only one transition

$$(a \longrightarrow P) \xrightarrow{a} P$$

So it is the process with one external choice with label a and continuation P , and empty internal and \checkmark -choices:

```
_ $\longrightarrow$ Str_ : {lu : LUniv}  $\rightarrow$  Label lu  $\rightarrow$  String  $\rightarrow$  String
l  $\longrightarrow$ Str s = "(" ++s showLabel l ++s "  $\rightarrow$  " ++s s ++s ")"
```



$$\begin{aligned}
 _ \longrightarrow + _ &: \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \\
 &\rightarrow \text{Process}\infty i c \rightarrow \text{Process}+ i c \\
 \text{E} \quad (l \longrightarrow + P) &= \top' \\
 \text{Lab} \quad (l \longrightarrow + P) \ c &= l \\
 \text{PE} \quad (l \longrightarrow + P) \ c &= P \\
 \text{I} \quad (l \longrightarrow + P) &= \emptyset' \\
 \text{PI} \quad (l \longrightarrow + P) \ () & \\
 \text{T} \quad (l \longrightarrow + P) &= \emptyset' \\
 \text{PT} \quad (l \longrightarrow + P) \ () & \\
 \text{Str}+ \quad (l \longrightarrow + P) &= l \longrightarrow \text{Str} \text{Str}\infty P
 \end{aligned}$$

$$\begin{aligned}
 _ \longrightarrow _ &: \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \\
 &\rightarrow \text{Process}\infty i c \rightarrow \text{Process} i c \\
 l \longrightarrow P &= \text{node } (l \longrightarrow + P)
 \end{aligned}$$

5.9 Internal Choice

The CSP offer the internal operator to leave the choice in the hands of the process. Here the environment does not have any authority to choose. We will here formalise internal choice operators. The internal choice operator has the following transitions:

$$P \sqcap Q \xrightarrow{\tau} P \qquad P \sqcap Q \xrightarrow{\tau} Q$$

It is modelled in CSP-Agda by having as internal choice set `bool` and otherwise empty choices:

$$\begin{aligned}
 _ \sqcap \text{Str} _ &: \text{String} \rightarrow \text{String} \rightarrow \text{String} \\
 s \sqcap \text{Str} s' &= "(" ++ s ++ s' " \sqcap " ++ s' ++ s ")"
 \end{aligned}$$

$$\begin{aligned}
 _ \sqcap + _ &: \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Process}\infty i \{lu\} c \\
 &\rightarrow \text{Process}\infty i \{lu\} c \rightarrow \text{Process}+ i \{lu\} c \\
 \text{E} \quad (P \sqcap + Q) &= \emptyset' \\
 \text{Lab} \quad (P \sqcap + Q) \ () & \\
 \text{PE} \quad (P \sqcap + Q) \ () &
 \end{aligned}$$


```

I    (P ⊔+ Q)           = fin 2
PI   (P ⊔+ Q) zero      = P
PI   (P ⊔+ Q) (suc zero) = Q
PI   (P ⊔+ Q) (suc (suc ()))
T    (P ⊔+ Q)           = ∅'
PT   (P ⊔+ Q) ()
Str+ (P ⊔+ Q)          = Str∞ P ⊔Str Str∞ Q

```

```

_⊔_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process i {lu} c
P ⊔ Q = node (P ⊔+ Q)

```

```

_⊔∞_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process∞ (↑ i) {lu} c
forcep (P ⊔∞ Q) {j} = P ⊔ Q
Str∞ (P ⊔∞ Q) = (Str∞ P) ⊔Str (Str∞ Q)

```

5.10 External Choice

External choice allows the environment to make the choice between the behaviour of the processes. For instance, the process $(a \longrightarrow P \sqcap b \longrightarrow Q)$ can engage in either of the events a or b . If the first event chosen was a , the posterior behaviour is described by P , and if it was b , the process will behave as Q . The inference rules for external choice are as follows (having an inference rule with two conclusions is an abbreviation for two inference rule, one deriving the first and one deriving the second conclusion):

$$\begin{array}{c}
\frac{P \xrightarrow{a} \bar{P}}{P \sqcap Q \xrightarrow{a} \bar{P}} \\
Q \sqcap P \xrightarrow{a} \bar{P}
\end{array}
\qquad
\begin{array}{c}
\frac{P \xrightarrow{\tau} \bar{P}}{P \sqcap Q \xrightarrow{\tau} \bar{P} \sqcap Q} \\
Q \sqcap P \xrightarrow{\tau} Q \sqcap \bar{P}
\end{array}$$

Assume processes $P : \text{Process } i \ c_0$ and $Q : \text{Process } i \ c_1$ and consider $P \sqcap Q$. If P or Q terminates, then $P \sqcap Q$ can terminate with the return value of that process. In case both processes are of the form `terminate` we need to be consistent with the behaviour we would have if both processes were `SKIP`: in that case the process could have two \checkmark -events corresponding to each of the two return values. So we get again return values in c_0 or c_1 . The result returned is therefore always in c_0 or c_1 , i.e. an element of the disjoint union

$(c_0 \uplus c_1)$ of c_0 and c_1 . In case P and Q have not terminated the defining equations are obvious from the rules. The only problem is that we have to map the return values of the processes ($\text{PE } P \ c$) to the return value of $P \sqcap Q$. We do this by using the function `fmap` defined below.

If both processes terminate, as said before we obtain a process, which can terminate with each of two given return values. So we obtain $(2\text{-}\checkmark \ a \ b)$ which is the process, which can make \checkmark transitions for return values $(\text{inj}_1 \ a)$ and $(\text{inj}_2 \ b)$. We would prefer to return $(\text{terminate} \ (a \ , \ b))$, but being consistent with that $(\text{terminate} \ a)$ should be semantically equal to $(\text{SKIP} \ a)$ requires this choice. Here $_ \ , \ _$ is the constructor of type $_ \times _$, which is defined in Agda as follows:

```
data _×_ (a b : Set) : Set where
  _,,_ : a → b → a × b
```

In case of $(\text{terminate} \ a \ \sqcap \ P)$ we get a more complex behaviour: (1) the combined process can terminate with result a ; (2) it can follow an internal choice of P , after which the possibility having a transition as in (1) remains; (3) we can have a termination event of P , in which case the result returned is that of P ; (4) we can have an external choice of P , in which case information about termination of the first process is lost. What we get is that the combined process behaves as P , but the return value needs to be mapped to the return value of the combined value. In addition, we need to add using `addTimed \checkmark` a timed tick event, which provides the possibility of having a transition $\xrightarrow{\checkmark, a}$, as long as the process hasn't performed an external choice operation. We obtain the following code:

```
_□Str_ : String → String → String
s □Str s' = "(" ++ s ++ s " □ " ++ s' ++ s ")"
```

mutual

```
_□∞∞_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
  → Process∞ i {lu} c0 → Process∞ i {lu} c1
  → Process∞ i {lu} (c0  $\uplus$  c1)
forcep (P □∞∞ Q) = forcep P □ forcep Q
Str∞ (P □∞∞ Q) = Str∞ P □Str Str∞ Q

_□∞+_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
  → Process∞ i {lu} c0 → Process+ i {lu} c1
```

$$\begin{aligned}
& \rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep } (P \sqcap\infty+ Q) &= \text{forcep } P \sqcap\text{p}+ Q \\
\text{Str}\infty (P \sqcap\infty+ Q) &= \text{Str}\infty P \sqcap\text{Str Str}+ Q \\
\\
_ \sqcap\infty+ _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process}+ i \{lu\} c_0 \rightarrow \text{Process}\infty i \{lu\} c_1 \\
&\rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep } (P \sqcap\infty+ Q) &= P \sqcap\text{p}+ \text{forcep } Q \\
\text{Str}\infty (P \sqcap\infty+ Q) &= \text{Str}+ P \sqcap\text{Str Str}\infty Q \\
\\
_ \sqcap_ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \rightarrow \text{Process } i \{lu\} c_0 \\
&\rightarrow \text{Process } i \{lu\} c_1 \rightarrow \text{Process } i \{lu\} (c_0 \uplus' c_1) \\
\text{node } P \sqcap Q &= P \sqcap\text{p}+ Q \\
P \sqcap \text{node } Q &= P \sqcap\text{p}+ Q \\
\text{terminate } a \sqcap \text{terminate } b &= 2\text{-}\checkmark a b \\
\\
_ \sqcap\text{p}+ _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process}+ i \{lu\} c_0 \rightarrow \text{Process } i \{lu\} c_1 \\
&\rightarrow \text{Process } i \{lu\} (c_0 \uplus' c_1) \\
P \sqcap\text{p}+ \text{terminate } b &= \text{addTimed}\checkmark (\text{inj}_2 b) (\text{node } (\text{fmap}+ \text{inj}_1 P)) \\
P \sqcap\text{p}+ \text{node } Q &= \text{node } (P \sqcap\text{p}+ Q) \\
\\
_ \sqcap\infty+ _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process}+ i \{lu\} c_0 \rightarrow \text{Process}\infty i \{lu\} c_1 \\
&\rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep } (P \sqcap\infty+ Q) &= \text{node } (P \sqcap\text{p}+ \text{forcep } Q) \\
\text{Str}\infty (P \sqcap\infty+ Q) &= \text{Str}+ P \sqcap\text{Str Str}\infty Q \\
\\
_ \sqcap\text{p}+ _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \rightarrow \text{Process } i \{lu\} c_0 \\
&\rightarrow \text{Process}+ i \{lu\} c_1 \rightarrow \text{Process } i \{lu\} (c_0 \uplus' c_1) \\
\text{terminate } a \sqcap\text{p}+ Q &= \text{addTimed}\checkmark (\text{inj}_1 a) \\
&\quad (\text{node } (\text{fmap}+ \text{inj}_2 Q)) \\
\text{node } P \sqcap\text{p}+ Q &= \text{node } (P \sqcap\text{p}+ Q) \\
\\
_ \sqcap\text{p}+ _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process}+ i \{lu\} c_0 \rightarrow \text{Process}+ i \{lu\} c_1 \\
&\rightarrow \text{Process}+ i \{lu\} (c_0 \uplus' c_1)
\end{aligned}$$

$$\begin{aligned}
E & (P \square+ Q) &= E P \uplus' E Q \\
\text{Lab} & (P \square+ Q) (\text{inj}_1 x) &= \text{Lab } P x \\
\text{Lab} & (P \square+ Q) (\text{inj}_2 x) &= \text{Lab } Q x \\
PE & (P \square+ Q) (\text{inj}_1 x) &= \text{fmap}\infty \text{ inj}_1 (PE P x) \\
PE & (P \square+ Q) (\text{inj}_2 x) &= \text{fmap}\infty \text{ inj}_2 (PE Q x) \\
I & (P \square+ Q) &= I P \uplus' I Q \\
PI & (P \square+ Q) (\text{inj}_1 c) &= PI P c \square\infty+ Q \\
PI & (P \square+ Q) (\text{inj}_2 c) &= P \square+\infty PI Q c \\
T & (P \square+ Q) &= T P \uplus' T Q \\
PT & (P \square+ Q) (\text{inj}_1 c) &= \text{inj}_1 (PT P c) \\
PT & (P \square+ Q) (\text{inj}_2 c) &= \text{inj}_2 (PT Q c) \\
\text{Str}+ & (P \square+ Q) &= \text{Str}+ P \square\text{Str} \text{Str}+ Q
\end{aligned}$$

$$\begin{aligned}
_ \square\infty p_ & : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\
& \rightarrow \text{Process}\infty i \{lu\} c_0 \rightarrow \text{Process } i \{lu\} c_1 \\
& \rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep} & (P \square\infty p Q) = \text{forcep } P \square Q \\
\text{Str}\infty & (P \square\infty p Q) = \text{Str}\infty P \square\text{Str} \text{Str} Q
\end{aligned}$$

$$\begin{aligned}
_ \square p\infty_ & : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\
& \rightarrow \text{Process } i \{lu\} c_0 \rightarrow \text{Process}\infty i \{lu\} c_1 \\
& \rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep} & (P \square p\infty Q) = P \square \text{forcep } Q \\
\text{Str}\infty & (P \square p\infty Q) = \text{Str } P \square\text{Str} \text{Str}\infty Q
\end{aligned}$$

We used here the function, which adds the possibility of terminating with result a , which is only available, as long as the process hasn't performed an external choice:

$$\begin{aligned}
\text{addTimed}\checkmark\text{Str} & : \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \\
& \rightarrow \text{String} \rightarrow \text{String} \\
\text{addTimed}\checkmark\text{Str } a \text{ str} &= "(\text{addTimed}\checkmark \text{ " ++s choice2Str } a \text{ ++s " } \\
& \quad \text{" ++s str ++s "})"
\end{aligned}$$

mutual

$$\text{addTimed}\checkmark\infty : \forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\}$$

```

      → Process $\infty$  i {lu} c → Process $\infty$  i {lu} c
forcep (addTimed $\checkmark$   $\infty$  a P) = addTimed $\checkmark$  a (forcep P)
Str $\infty$  (addTimed $\checkmark$   $\infty$  a P) = addTimed $\checkmark$  Str a (Str $\infty$  P)

addTimed $\checkmark$  :  $\forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\}$ 
      → Process i {lu} c → Process i {lu} c
addTimed $\checkmark$  a (terminate b) = fmap unifyA $\oplus$ A (2- $\checkmark$  a b)
addTimed $\checkmark$  a (node P) = node (addTimed $\checkmark$  + a P)

```

```

addTimed $\checkmark$  + :  $\forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\}$ 
      → Process+ i {lu} c
      → Process+ i {lu} c
E (addTimed $\checkmark$  + a P) = E P
Lab (addTimed $\checkmark$  + a P) = Lab P
PE (addTimed $\checkmark$  + a P) s = PE P s
I (addTimed $\checkmark$  + a P) = I P
PI (addTimed $\checkmark$  + a P) s = addTimed $\checkmark$   $\infty$  a (PI P s)
T (addTimed $\checkmark$  + a P) = T'  $\oplus$  T P
PT (addTimed $\checkmark$  + a P) (inj1 _) = a
PT (addTimed $\checkmark$  + a P) (inj2 c) = PT P c
Str+ (addTimed $\checkmark$  + a P) = addTimed $\checkmark$  Str a (Str+ P)

```

The process having two tick events for two values is defined as follows:

```

2- $\checkmark$ Str : {c0 c1 : Choice} → (a : ChoiceSet c0)
      → (b : ChoiceSet c1) → String
2- $\checkmark$ Str a b = "(2- $\checkmark$  " ++s choice2Str a ++s " " ++s choice2Str b ++s ")"

2- $\checkmark$  + :  $\forall \{i\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c_0)$ 
      → {lu : LUniv}
      → (b : ChoiceSet c1) → Process+ i {lu} (c0  $\oplus$  c1)
E (2- $\checkmark$  + a b) =  $\emptyset$ '
Lab (2- $\checkmark$  + a b) = ()
PE (2- $\checkmark$  + a b) = ()
I (2- $\checkmark$  + a b) =  $\emptyset$ '
PI (2- $\checkmark$  + a b) = ()
T (2- $\checkmark$  + a b) = fin 2
PT (2- $\checkmark$  + a b) zero = inj1 a
PT (2- $\checkmark$  + a b) (suc zero) = inj2 b

```

```

PT (2-✓+ a b) (suc (suc ()))
Str+ (2-✓+ a b) = "(2-✓ " ++s choice2Str a ++s "
                  " ++s choice2Str b ++s ")"

```

```

2-✓ : ∀ {i} → {c0 c1 : Choice} → (a : ChoiceSet c0) → {lu : LUniv}
      → (b : ChoiceSet c1) → Process i {lu} (c0 ⊕' c1)

```

```

2-✓ a b = node (2-✓+ a b)

```

```

2-✓∞ : ∀ {i} → {c0 c1 : Choice} → (a : ChoiceSet c0) → {lu : LUniv}
      → (b : ChoiceSet c1) → Process∞ i {lu} (c0 ⊕' c1)
forcep (2-✓∞ a b) = (2-✓ a b)
Str∞ (2-✓∞ a b) = 2-✓Str a b

```

The function `fmap` mapping $(\text{Process } i \ c_0)$ to $(\text{Process } i \ c_1)$ by applying a function $(f : \text{ChoiceSet } c_0 \rightarrow \text{ChoiceSet } c_1)$ to the return values can be defined corecursively as follows:

```

fmapStr : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
      → String → String
fmapStr f str = "(fmap " ++s choiceFunToStr↓ f ++s " " ++s str ++s ")"

```

mutual

```

fmap∞ : {c0 c1 : Choice}
      → (f : ChoiceSet c0 → ChoiceSet c1)
      → {i : Size}
      → {lu : LUniv}
      → Process∞ i {lu} c0 → Process∞ i {lu} c1
forcep (fmap∞ f P) = fmap f (forcep P)
Str∞ (fmap∞ f P) = fmapStr f (Str∞ P)

```

```

fmap : {lu : LUniv}{c0 c1 : Choice} → (f : ChoiceSet c0
      → ChoiceSet c1) → {i : Size}
      → Process i {lu} c0 → Process i {lu} c1
fmap f (terminate a) = terminate (f a)
fmap f (node P)      = node (fmap+ f P)

```

```

fmap+ : {c0 c1 : Choice}
       → (f : ChoiceSet c0 → ChoiceSet c1)
       → {i : Size}
       → {lu : LUniv}
       → Process+ i {lu} c0 → Process+ i {lu} c1
E   (fmap+ f P) = E P
Lab (fmap+ f P) c = Lab P c
PE  (fmap+ f P) c = fmap∞ f (PE P c)
I   (fmap+ f P) = I P
PI  (fmap+ f P) c = fmap∞ f (PI P c)
T   (fmap+ f P) = T P
PT  (fmap+ f P) c = f (PT P c)
Str+ (fmap+ f P) = fmapStr f (Str+ P)

```

5.11 Renaming

The renaming operator takes a process and renames the external choice labels by applying a function to them. It is modelled in CSP-Agda as follows:

```

RenameStr : {lu : LUniv} (f : Label lu → Label lu) → String → String
RenameStr f s = "(" ++s s ++s ")" ++s (labelLabelFunToString f)

```

mutual

```

Rename∞ : {i : Size} → {lu : LUniv} {c : Choice}
         → (f : Label lu → Label lu)
         → Process∞ i {lu} c → Process∞ i {lu} c
forcep (Rename∞ f P) = Rename f (forcep P)
Str∞ (Rename∞ f P) = RenameStr f (Str∞ P)

Rename : {i : Size} → {lu : LUniv} {c : Choice}
        → (f : Label lu → Label lu)
        → Process i {lu} c → Process i {lu} c
Rename f (node P) = node (Rename+ f P)
Rename f (terminate x) = terminate x

Rename+ : {i : Size} → {lu : LUniv} {c : Choice}
         → (f : Label lu → Label lu)

```

$$\begin{aligned}
& \rightarrow \text{Process+ } i \{lu\} c \rightarrow \text{Process+ } i \{lu\} c \\
\text{E } (\text{Rename+ } f P) &= (\text{E } P) \\
\text{Lab } (\text{Rename+ } f P) c &= f (\text{Lab } P c) \\
\text{PE } (\text{Rename+ } f P) c &= \text{Rename}\infty f (\text{PE } P c) \\
\text{I } (\text{Rename+ } f P) &= \text{I } P \\
\text{PI } (\text{Rename+ } f P) c &= \text{Rename}\infty f (\text{PI } P c) \\
\text{T } (\text{Rename+ } f P) &= \text{T } P \\
\text{PT } (\text{Rename+ } f P) c &= \text{PT } P c \\
\text{Str+ } (\text{Rename+ } f P) &= \text{RenameStr } f (\text{Str+ } P)
\end{aligned}$$

5.12 Hiding

Hiding allows to hide some external transitions and replace them by silent ones in order to hide them from other processes. The behaviour of the hiding operator is shown by the following firing rules:

$$\frac{P \xrightarrow{a} \bar{P}}{P \setminus A \xrightarrow{\tau} \bar{P} \setminus A} [a \in A] \quad \frac{P \xrightarrow{\mu} \bar{P}}{P \setminus A \xrightarrow{\mu} \bar{P} \setminus A} [\mu \notin A]$$

In our approach we model this operator as follows (the parameter *hide* determines whether a label is hidden or not):

HideStr : $\{lu : \text{LUniv}\} (f : \text{Label } lu \rightarrow \text{Bool}) \rightarrow \text{String} \rightarrow \text{String}$
HideStr $f \text{ str} = \text{"Hide " ++ labelBoolFunToString } f \text{ ++ " " ++ str}$

mutual

Hide ∞ : $\{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c : \text{Choice}\}$
 $\rightarrow (hide : \text{Label } lu \rightarrow \text{Bool})$
 $\rightarrow \text{Process}\infty i \{lu\} c$
 $\rightarrow \text{Process}\infty i \{lu\} c$
forcep (**Hide** ∞ f P) = **Hide** f (**forcep** P)
Str ∞ (**Hide** ∞ f P) = **HideStr** f (**Str** ∞ P)

Hide : $\{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c : \text{Choice}\}$
 $\rightarrow (hide : \text{Label } lu \rightarrow \text{Bool})$
 $\rightarrow \text{Process } i \{lu\} c$
 $\rightarrow \text{Process } i \{lu\} c$
Hide f (**node** P) = **node** (**Hide**+ f P)
Hide f (**terminate** x) = **terminate** x

```

Hide+ : {i : Size} → {lu : LUniv} → {c : Choice}
       → (hide : Label lu → Bool) → Process+ i {lu} c
       → Process+ i {lu} c
E   (Hide+ f P) = subset' (E P) (¬b ∘ (f ∘ (Lab P)))
Lab (Hide+ f P) c = Lab P (projSubset c)
PE  (Hide+ f P) c = Hide∞ f (PE P (projSubset c))
I   (Hide+ f P) = I P ⊕' subset' (E P) (f ∘ Lab P)
PI  (Hide+ f P) (inj1 c) = Hide∞ f (PI P c)
PI  (Hide+ f P) (inj2 c) = Hide∞ f (PE P (projSubset c))
T   (Hide+ f P) = T P
PT  (Hide+ f P) = PT P
Str+ (Hide+ f P) = HideStr f (Str+ P)

```

Here $\neg b$ is Boolean negation. In our approach the external choice $E P$ is the subset of the external choices for which $Lab P$ is not hidden, and the internal choice $I P$ is the union of the internal choice and the subset of the external choice for which $Lab P$ is hidden.

5.13 Interleaving Operator

The interleaving operator executes the external and internal choices of its arguments P and Q completely independently of each other. The CSP rules are as follows:

$$\begin{array}{c}
 \frac{P \xrightarrow{\checkmark} \bar{P} \quad Q \xrightarrow{\checkmark} \bar{Q}}{P ||| Q \xrightarrow{\checkmark} \bar{P} ||| \bar{Q}} \quad \frac{P \xrightarrow{\mu} \bar{P}}{P ||| Q \xrightarrow{\mu} \bar{P} ||| Q} \mu \neq \checkmark \\
 \quad \quad \quad Q ||| P \xrightarrow{\mu} Q ||| \bar{P}
 \end{array}$$

The definition in CSP-Agda is as follows:

```

_|||Str_ : String → String → String
s |||Str s' = s ++s "|||" ++s s'

```

```

mutual
  _|||∞_ : {i : Size}{lu : LUniv}
        → {c0 c1 : Choice}
        → Process∞ i {lu} c0
        → Process∞ i {lu} c1

```

$$\begin{aligned}
& \rightarrow \text{Process}_{\infty} i \{lu\} (c_0 \times' c_1) \\
\text{forcep} (P \parallel_{\infty} Q) &= \text{forcep } P \parallel \text{forcep } Q \\
\text{Str}_{\infty} (P \parallel_{\infty} Q) &= \text{Str}_{\infty} P \parallel \text{Str } \text{Str}_{\infty} Q \\
\\
-|||_{} : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process } i \{lu\} \ c_0 \\
&\rightarrow \text{Process } i \ c_1 \rightarrow \text{Process } i \{lu\} (c_0 \times' c_1) \\
\text{node } P \parallel \text{node } Q &= \text{node } (P \parallel ++ Q) \\
\text{terminate } a \parallel Q &= \text{fmap } (\lambda b \rightarrow (a \text{ ,, } b)) \ Q \\
P \parallel \text{terminate } b &= \text{fmap } (\lambda a \rightarrow (a \text{ ,, } b)) \ P \\
\\
-|||_{\infty}+ : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}_{\infty} i \{lu\} \ c_0 \\
&\rightarrow \text{Process}_{+} i \{lu\} \ c_1 \rightarrow \text{Process}_{\infty} i \{lu\} (c_0 \times' c_1) \\
\text{forcep} (P \parallel_{\infty}+ Q) &= \text{node } (\text{forcep } P \parallel \text{p+ } Q) \\
\text{Str}_{\infty} (P \parallel_{\infty}+ Q) &= \text{Str}_{\infty} P \parallel \text{Str } \text{Str}_{+} Q \\
\\
-|||_{+} : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}_{+} i \{lu\} \ c_0 \\
&\rightarrow \text{Process}_{\infty} i \{lu\} \ c_1 \rightarrow \text{Process}_{\infty} i \{lu\} (c_0 \times' c_1) \\
\text{forcep} (P \parallel_{+} Q) &= \text{node } (P \parallel \text{p+ } \text{forcep } Q) \\
\text{Str}_{\infty} (P \parallel_{+} Q) &= \text{Str}_{+} P \parallel \text{Str } \text{Str}_{\infty} Q \\
\\
-|||_{\text{p+}} : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process } i \{lu\} \ c_0 \\
&\rightarrow \text{Process}_{+} i \{lu\} \ c_1 \rightarrow \text{Process}_{+} i \{lu\} (c_0 \times' c_1) \\
\text{terminate } a \parallel \text{p+ } Q &= \text{fmap}_{+} (\lambda b \rightarrow (a \text{ ,, } b)) \ Q \\
\text{node } P \parallel \text{p+ } Q &= P \parallel ++ Q \\
\\
-|||_{\text{p+}} : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}_{+} i \ c_0 \\
&\rightarrow \text{Process } i \{lu\} \ c_1 \rightarrow \text{Process}_{+} i \{lu\} (c_0 \times' c_1) \\
P \parallel \text{p+ } \text{terminate } b &= \text{fmap}_{+} (\lambda a \rightarrow (a \text{ ,, } b)) \ P \\
P \parallel \text{p+ } \text{node } Q &= P \parallel ++ Q \\
\\
-|||_{++} : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \\
&\rightarrow \text{Process}_{+} i \{lu\} \ c_0 \rightarrow \text{Process}_{+} i \{lu\} \ c_1 \\
&\rightarrow \text{Process}_{+} i \{lu\} (c_0 \times' c_1) \\
\text{E } (P \parallel ++ Q) &= \text{E } P \uplus' \text{E } Q \\
\text{Lab } (P \parallel ++ Q) (\text{inj}_1 \ c) &= \text{Lab } P \ c \\
\text{Lab } (P \parallel ++ Q) (\text{inj}_2 \ c) &= \text{Lab } Q \ c \\
\text{PE } (P \parallel ++ Q) (\text{inj}_1 \ c) &= \text{PE } P \ c \parallel_{\infty}+ Q \\
\text{PE } (P \parallel ++ Q) (\text{inj}_2 \ c) &= P \parallel_{+} \text{PE } Q \ c \\
\text{I } (P \parallel ++ Q) &= \text{I } P \uplus' \text{I } Q \\
\text{PI } (P \parallel ++ Q) (\text{inj}_1 \ c) &= \text{PI } P \ c \parallel_{\infty}+ Q
\end{aligned}$$

$$\begin{aligned}
\text{PI} \quad (P \parallel\!\!\parallel + Q) (\text{inj}_2 \, c) &= P \parallel\!\!\parallel +^\infty \text{PI} \, Q \, c \\
\text{T} \quad (P \parallel\!\!\parallel + Q) &= \text{T} \, P \times' \text{T} \, Q \\
\text{PT} \quad (P \parallel\!\!\parallel + Q) (c \, ,, \, c_1) &= \text{PT} \, P \, c \, ,, \, \text{PT} \, Q \, c_1 \\
\text{Str}+ \quad (P \parallel\!\!\parallel + Q) &= \text{Str}+ \, P \parallel\!\!\parallel \text{Str} \, \text{Str}+ \, Q
\end{aligned}$$

$$\begin{aligned}
- \parallel\!\!\parallel \text{p}\infty - : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \, c_1 : \text{Choice}\} \rightarrow \text{Process} \, i \, \{lu\} \, c_0 \\
&\rightarrow \text{Process}\infty \, i \, \{lu\} \, c_1 \rightarrow \text{Process}\infty \, i \, \{lu\} \, (c_0 \times' c_1) \\
\text{forcep} \quad (P \parallel\!\!\parallel \text{p}\infty \, Q) &= P \parallel\!\!\parallel \text{forcep} \, Q \\
\text{Str}\infty \quad (P \parallel\!\!\parallel \text{p}\infty \, Q) &= \text{Str} \, P \parallel\!\!\parallel \text{Str} \, \text{Str}\infty \, Q
\end{aligned}$$

$$\begin{aligned}
- \parallel\!\!\parallel \infty \text{p} - : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \, c_1 : \text{Choice}\} \rightarrow \text{Process}\infty \, i \, \{lu\} \, c_0 \\
&\rightarrow \text{Process} \, i \, \{lu\} \, c_1 \rightarrow \text{Process}\infty \, i \, \{lu\} \, (c_0 \times' c_1) \\
\text{forcep} \quad (P \parallel\!\!\parallel \infty \text{p} \, Q) &= \text{forcep} \, P \parallel\!\!\parallel Q \\
\text{Str}\infty \quad (P \parallel\!\!\parallel \infty \text{p} \, Q) &= \text{Str}\infty \, P \parallel\!\!\parallel \text{Str} \, \text{Str} \, Q
\end{aligned}$$

When processes P and Q haven't terminated, then $P \parallel\!\!\parallel Q$ will not terminate. The external choices are the external choices of P and Q . The labels are the labels from the processes P and Q , and we continue recursively with the interleaving combination. The internal choices are defined similarly. A termination event can happen only if both processes have a termination event.

If one process terminates but the other not, the rules of CSP express that one continues as the other process, until it has terminated. We can therefore equate, if P has terminated, $P \parallel\!\!\parallel Q$ with Q . However, we record the result obtained by P , and therefore apply fmap to Q in order to add the result of P to the result of Q when it terminates. If both processes terminate with results a and b , then the interleaving combination terminates with result $(a \, ,, \, b)$.

5.14 The Parallel Operator

The parallel operator gives the possibility to enforce two processes to work together and interact through synchronous events. For each of the two processes sets of labels A, B are given. For labels, which are not in the intersection, both processes can execute independently, as long as their processes are in A or B , respectively. For labels in the intersection, both processes need to synchronise on that event. The transition rules for the parallel operator are as follows:

$$\begin{array}{c}
\frac{P \xrightarrow{a} \bar{P} \quad Q \xrightarrow{a} \bar{Q}}{P \parallel [A \mid B] \parallel Q \xrightarrow{a} \bar{P} \parallel [A \mid B] \parallel \bar{Q}} [a \in A \cup \{\checkmark\} \cap B \cup \{\checkmark\}] \\
\\
\frac{P \xrightarrow{\mu} \bar{P}}{P \parallel [A \mid B] \parallel Q \xrightarrow{\mu} \bar{P} \parallel [A \mid B] \parallel Q} [\mu \in ((A \cup \tau) \setminus B)] \\
\\
Q \parallel [A \mid B] \parallel P \xrightarrow{\mu} Q \parallel [A \mid B] \parallel \bar{P}
\end{array}$$

In CSP-Agda we define the parallel operator as follows: We assume functions $A \ B : \text{Label} \rightarrow \text{Bool}$ which determine the label sets A and B as above. The external choices of $P \parallel [A \mid B] \parallel Q$ are:

- The external choices of $c : \text{E } P$, for which the label in P is in $(A \setminus B)$, i.e. such that $((A \setminus B) (\text{Lab } P \ c)) = \text{true}$. Here $(A \setminus B) : \text{Label} \rightarrow \text{Bool}$ is defined by $(A \setminus B) \ b = \text{true}$ if and only if $A \ b = \text{true}$ and $B \ b = \text{false}$. For such c the label for this external choice is the label of P for choice c , and the process obtained following this transition is the parallel construct applied to $\text{PE } P \ c$ and Q .
- The external choices of $c : \text{E } Q$, for which the label in Q is in $(B \setminus A)$, with similar definitions of the label and next process obtained.
- The combined external choices for P and Q , i.e. pairs (e_1, e_2) s.t. $e_1 : \text{E } P$ and $e_2 : \text{E } Q$, and s.t. their labels are equal, and the labels are in sets A and in B respectively, i.e.

$$(\text{Lab } P \ e_1 == \text{Lab } Q \ e_2) \wedge A (\text{Lab } P \ e_1) \wedge B (\text{Lab } Q \ e_2) = \text{true}$$

Here $==$ is Boolean valued equality on Labels, and \wedge is Boolean valued conjunction. The label for this external choice is the label of P (which is with respect to $==$ equal to the corresponding label of Q). The process obtained when following this external choice is the parallel construct applied to the result of following the external choices in both P and Q .

Furthermore

- The internal choices are the internal choices of P and Q , and the process obtained when following those transitions is obtained by following the corresponding transition in process P or Q , respectively.

- A termination event can happen only if both processes have a termination event. If they terminate with results a and b , then the parallel combination terminates with result $(a \mathbin{..} b)$. Therefore the result type of the parallel construct is the product of the result type of the first and second process.

In order to define the above we use the `subset'` constructor of `Choice`, which has equality rule

$$\text{ChoiceSet } (\text{subset}' E f) = \text{subset } (\text{ChoiceSet } E) f$$

Here `subset` $a f$ is the set of pairs `sub` $a b$ such that $a : A$ and $b : \top (f a)$, i.e. it is essentially the set $\{a : A \mid f a = \text{true}\}$. We have $\top : \text{Bool} \rightarrow \text{Set}$, such that $(\top \text{true})$ is provable and $(\top \text{false})$ is empty, i.e. not provable.

The definition of the parallel operator in CSP-Agda is as follows:

mutual

$$\begin{aligned} -[] \parallel \infty [] - & : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ & \rightarrow \text{Process} \infty i \{lu\} c_0 \\ & \rightarrow (A \ B : \text{Label } lu \rightarrow \text{Bool}) \\ & \rightarrow \text{Process} \infty i \{lu\} c_1 \\ & \rightarrow \text{Process} \infty i \{lu\} (c_0 \times' c_1) \\ \text{forcep } (P [A] \parallel \infty [B] Q) & = \text{forcep } P [A] \parallel [B] \text{forcep } Q \\ \text{Str} \infty (P [A] \parallel \infty [B] Q) & = \text{Str} \infty P [A] \parallel \text{Str} [B] \text{Str} \infty Q \end{aligned}$$

$$\begin{aligned} -[] \parallel + [] - & : \{lu : \text{LUniv}\} \{i : \text{Size}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ & \rightarrow \text{Process} + i \{lu\} c_0 \\ & \rightarrow (A \ B : \text{Label } lu \rightarrow \text{Bool}) \\ & \rightarrow \text{Process} + i \{lu\} c_1 \\ & \rightarrow \text{Process} + i \{lu\} (c_0 \times' c_1) \\ \text{E } (P [A] \parallel + [B] Q) & = \text{subset}' (\text{E } P) ((A \setminus B) \circ (\text{Lab } P)) \uplus' \\ & \quad \text{subset}' (\text{E } Q) ((B \setminus A) \circ (\text{Lab } Q)) \uplus' \\ & \quad \text{subset}' (\text{E } P \times' \text{E } Q) \\ & \quad (\lambda \{(e_1 \mathbin{..} e_2)\} \\ & \quad \rightarrow \text{Lab } P e_1 == \text{Lab } Q e_2 \wedge A (\text{Lab } P e_1) \wedge B (\text{Lab } Q e_2)) \\ \text{Lab } (P [A] \parallel + [B] Q) (\text{inj}_1 (\text{inj}_1 (\text{sub } c p))) & = \text{Lab } P c \\ \text{Lab } (P [A] \parallel + [B] Q) (\text{inj}_1 (\text{inj}_2 (\text{sub } c p))) & = \text{Lab } Q c \\ \text{Lab } (P [A] \parallel + [B] Q) (\text{inj}_2 (\text{sub } (c_0 \mathbin{..} c_1) p)) & = \text{Lab } P c_0 \\ \text{PE } (P [A] \parallel + [B] Q) (\text{inj}_1 (\text{inj}_1 (\text{sub } c p))) & = \text{PE } P c [A] \parallel \infty + [B] Q \\ \text{PE } (P [A] \parallel + [B] Q) (\text{inj}_1 (\text{inj}_2 (\text{sub } c p))) & = P [A] \parallel + \infty [B] \text{PE } Q c \\ \text{PE } (P [A] \parallel + [B] Q) (\text{inj}_2 (\text{sub } (c_0 \mathbin{..} c_1) p)) & = \text{PE } P c_0 [A] \parallel \infty [B] \text{PE } Q c_1 \end{aligned}$$

$$\begin{array}{ll}
\mathbf{I} & (P \parallel [A] \parallel [B] Q) = \mathbf{I} P \uplus' \mathbf{I} Q \\
\mathbf{PI} & (P \parallel [A] \parallel [B] Q) (\text{inj}_1 c) = \mathbf{PI} P c \parallel [A] \parallel \infty [B] Q \\
\mathbf{PI} & (P \parallel [A] \parallel [B] Q) (\text{inj}_2 c) = P \parallel [A] \parallel \infty [B] \mathbf{PI} Q c \\
\mathbf{T} & (P \parallel [A] \parallel [B] Q) = \mathbf{T} P \times' \mathbf{T} Q \\
\mathbf{PT} & (P \parallel [A] \parallel [B] Q) (c_0 \text{ ,, } c_1) = (\mathbf{PT} P c_0 \text{ ,, } \mathbf{PT} Q c_1) \\
\mathbf{Str+} & (P \parallel [A] \parallel [B] Q) = \mathbf{Str+} P \parallel [A] \parallel \mathbf{Str}[B] \mathbf{Str+} Q
\end{array}$$

When defining the parallel construct for elements of **Process**, we need to deal with the case that one of the processes is the terminated process. As for \parallel , one continues in this case as the other other process, until it has terminated. However, in case of P having terminated, only labels in the set $(B \setminus A)$ are allowed for Q . We can therefore equate, if P has terminated, $P \parallel [A] \parallel [B] Q$ with $Q \upharpoonright (B \setminus A)$. Here for a process P' and a set of labels A' the process $P \upharpoonright A'$ is the process obtained by restricting the external transitions to those with label in A' . It is defined in Agda as follows:

```

 $\_ \upharpoonright \mathbf{Str} \_ : \{lu : \mathbf{LUniv}\} \rightarrow \mathbf{String} \rightarrow (A : \mathbf{Label} \text{ } lu \rightarrow \mathbf{Bool}) \rightarrow \mathbf{String}$ 
str  $\upharpoonright \mathbf{Str} A = \text{"Restrict " ++ s labelBoolFunToString A ++ s " " ++ s str}$ 

```

mutual

```

 $\_ \upharpoonright \infty \_ : \{lu : \mathbf{LUniv}\} \{i : \mathbf{Size}\} \rightarrow \{c : \mathbf{Choice}\} \rightarrow \mathbf{Process} \infty i \{lu\} c$ 
 $\rightarrow (A : \mathbf{Label} \text{ } lu \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Process} \infty i \{lu\} c$ 
forcep  $(P \upharpoonright \infty A) = (\text{forcep } P) \upharpoonright A$ 
Str $\infty (P \upharpoonright \infty A) = (\mathbf{Str} \infty P) \upharpoonright \mathbf{Str} A$ 

 $\_ \upharpoonright \_ : \{lu : \mathbf{LUniv}\} \{i : \mathbf{Size}\} \rightarrow \{c : \mathbf{Choice}\} \rightarrow \mathbf{Process} i \{lu\} c$ 
 $\rightarrow (A : \mathbf{Label} \text{ } lu \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Process} i \{lu\} c$ 
terminate  $a \upharpoonright A = \text{terminate } a$ 
node  $P \upharpoonright A = \text{node } (P \upharpoonright + A)$ 

```

```

 $\_ \upharpoonright + \_ : \{lu : \mathbf{LUniv}\} \{i : \mathbf{Size}\} \rightarrow \{c : \mathbf{Choice}\} \rightarrow \mathbf{Process+} i \{lu\} c$ 
 $\rightarrow (A : \mathbf{Label} \text{ } lu \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Process+} i \{lu\} c$ 
E  $(P \upharpoonright + A) = \text{subset'} (E P) (A \circ (\mathbf{Lab} P))$ 
Lab  $(P \upharpoonright + A) (\text{sub } c p) = \mathbf{Lab} P c$ 
PE  $(P \upharpoonright + A) (\text{sub } c p) = \mathbf{PE} P c \upharpoonright \infty A$ 
I  $(P \upharpoonright + A) = \mathbf{I} P$ 
PI  $(P \upharpoonright + A) c = \mathbf{PI} P c \upharpoonright \infty A$ 
T  $(P \upharpoonright + A) = \mathbf{T} P$ 
PT  $(P \upharpoonright + A) c = \mathbf{PT} P c$ 

```

$$\text{Str+ } (P \upharpoonright + A) = \text{Str+ } P \upharpoonright \text{Str } A$$

Note that this is different from hiding: external transitions with labels not in A' turned into τ -transitions. As for $_||_,$ we need to record the result obtained by P , and therefore apply `fmap` to Q in order to add the result of P to the result of the restriction of Q , when it terminates.

The definition of the parallel operator for `Process` is therefore as follows:

$$\begin{aligned} _||_ : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ &\rightarrow \text{Process } i \ \{lu\} \ c_0 \\ &\rightarrow (A \ B : \text{Label } lu \rightarrow \text{Bool}) \\ &\rightarrow \text{Process } i \ \{lu\} \ c_1 \\ &\rightarrow \text{Process } i \ \{lu\} \ (c_0 \times' \ c_1) \\ \text{node } P \ [\ A \]||\ [\ B \] \ \text{node } Q &= \text{node } (P \ [\ A \]||+ [\ B \] \ Q) \\ \text{terminate } a \ [\ A \]||\ [\ B \] \ Q &= \text{fmap } (\lambda \ b \rightarrow (a \ _, \ b)) \ (Q \upharpoonright (B \setminus A)) \\ P \ [\ A \]||\ [\ B \] \ \text{terminate } b &= \text{fmap } (\lambda \ a \rightarrow (a \ _, \ b)) \ (P \upharpoonright (A \setminus B)) \end{aligned}$$

5.15 Interrupt Operator

The interrupt operator passes control from one process to another one at an arbitrary point of execution. This means that transitions in the first process are just performed in the first argument of the interrupt operator, whereas if the second process has an external transition, then the combined process exits to the result of that transition. If one of the two processes terminates the combined process terminates. As for external choice, in case both processes terminate we need to return a process, which can have two tick events to the two results. So the return value is $(c_0 \uplus' c_1)$. The CSP rules are as follows:

$$\begin{array}{c} \frac{P \xrightarrow{\mu} \bar{P}}{P \triangle Q \xrightarrow{\mu} \bar{P} \triangle Q} \ [\mu \neq \checkmark] \quad \frac{P \xrightarrow{\checkmark} \bar{P}}{P \triangle Q \xrightarrow{\checkmark} \bar{P}} \\[10pt] \frac{Q \xrightarrow{\tau} \bar{Q}}{P \triangle Q \xrightarrow{\tau} P \triangle \bar{Q}} \quad \frac{Q \xrightarrow{a} \bar{Q}}{P \triangle Q \xrightarrow{a} \bar{Q}} \end{array}$$

CSP-Agda models the interrupt operator as follows:

mutual

$$_ \triangle \infty \infty _ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\}$$

$$\begin{aligned}
& \rightarrow \text{Process}\infty i \{lu\} c_0 \rightarrow \text{Process}\infty i \{lu\} c_1 \\
& \rightarrow \text{Process}\infty i \{lu\} (c_0 \uplus' c_1) \\
\text{forcep} (P \triangle\infty\infty P') &= \text{forcep} P \triangle \text{forcep} P' \\
\text{Str}\infty (P \triangle\infty\infty P') &= \text{Str}\infty P \triangle \text{Str}\infty P'
\end{aligned}$$

$$\begin{aligned}
_ \triangle _ : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process} i \{lu\} c_0 \rightarrow \text{Process} i \{lu\} c_1 \\
&\rightarrow \text{Process} i \{lu\} (c_0 \uplus' c_1) \\
\text{node } P \triangle P' &= P \triangle + p P' \\
P \triangle \text{node } P' &= P \triangle p + P' \\
\text{terminate } a \triangle \text{terminate } b &= 2-\checkmark a b
\end{aligned}$$

$$\begin{aligned}
_ \triangle + _ : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} &\rightarrow \{i : \text{Size}\} \\
&\rightarrow \text{Process}+ i \{lu\} c_0 \rightarrow \text{Process}+ i \{lu\} c_1 \\
&\rightarrow \text{Process}+ i \{lu\} (c_0 \uplus' c_1) \\
E (P \triangle + Q) &= E P \uplus' E Q \\
\text{Lab} (P \triangle + Q) (\text{inj}_1 x) &= \text{Lab} P x \\
\text{Lab} (P \triangle + Q) (\text{inj}_2 x) &= \text{Lab} Q x \\
PE (P \triangle + Q) (\text{inj}_1 x) &= PE P x \triangle \infty + Q \\
PE (P \triangle + Q) (\text{inj}_2 x) &= \text{fmap}\infty \text{inj}_2 (PE Q x) \\
I (P \triangle + Q) &= I P \uplus' I Q \\
PI (P \triangle + Q) (\text{inj}_1 c) &= PI P c \triangle \infty + Q \\
PI (P \triangle + Q) (\text{inj}_2 c) &= P \triangle + \infty PI Q c \\
T (P \triangle + Q) &= T P \uplus' T Q \\
PT (P \triangle + Q) (\text{inj}_1 c) &= \text{inj}_1 (PT P c) \\
PT (P \triangle + Q) (\text{inj}_2 c) &= \text{inj}_2 (PT Q c) \\
\text{Str}+ (P \triangle + Q) &= \text{Str}+ P \triangle \text{Str}+ Q
\end{aligned}$$

As before the rules in case none of the processes is (`terminate a`) are straightforward. If the first process is (`terminate a`) we see that the combined process operates liked the second process, but with the return value remapped, and with the addition of adding a typed tick as long as it performs no external choice. If the second process is (`terminate a`) the combined operator operates like the first process, with the same operations but replacing a timed tick by a permanently added tick. If both processes terminate then we get as result as for external choice a process (`2-✓ab`) having tick events for both return values.

The type of the function `add✓` with the clause, which differs from `addTimed✓`

is as follows:

mutual

$$\text{add}\checkmark\infty : \forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\} \\ \rightarrow \text{Process}\infty i \{lu\} c \rightarrow \text{Process}\infty i \{lu\} c$$

$$\text{forcep} (\text{add}\checkmark\infty a P) = \text{add}\checkmark a (\text{forcep } P) \\ \text{Str}\infty (\text{add}\checkmark\infty a P) = \text{add}\checkmark\text{Str } a (\text{Str}\infty P)$$

$$\text{add}\checkmark : \forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\} \\ \rightarrow \text{Process } i \{lu\} c \rightarrow \text{Process } i \{lu\} c \\ \text{add}\checkmark a (\text{terminate } b) = \text{fmap unifyA}\oplus\text{A } (2\text{-}\checkmark a b) \\ \text{add}\checkmark a (\text{node } P) = \text{node } (\text{add}\checkmark+ a P)$$

$$\text{add}\checkmark+ : \forall \{i\} \rightarrow \{c : \text{Choice}\} \rightarrow (a : \text{ChoiceSet } c) \rightarrow \{lu : \text{LUniv}\} \\ \rightarrow \text{Process}+ i \{lu\} c \rightarrow \text{Process}+ i \{lu\} c \\ \text{E } (\text{add}\checkmark+ a P) = \text{E } P \\ \text{Lab } (\text{add}\checkmark+ a P) = \text{Lab } P \\ \text{PE } (\text{add}\checkmark+ a P) s = \text{add}\checkmark\infty a (\text{PE } P s) \\ \text{I } (\text{add}\checkmark+ a P) = \text{I } P \\ \text{PI } (\text{add}\checkmark+ a P) s = \text{add}\checkmark\infty a (\text{PI } P s) \\ \text{T } (\text{add}\checkmark+ a P) = \text{T}' \oplus' \text{T } P \\ \text{PT } (\text{add}\checkmark+ a P) (\text{inj}_1 _) = a \\ \text{PT } (\text{add}\checkmark+ a P) (\text{inj}_2 c) = \text{PT } P c \\ \text{Str}+ (\text{add}\checkmark+ a P) = \text{add}\checkmark\text{Str } a (\text{Str}+ P)$$

In this chapter, we showed how to give the type theoretic interactive theorem prover Agda the ability to model concurrent programs by representing the process algebra CSP in monadic form. The operators of CSP are defined operations, which combine processes defined from atomic operations. In the next Chapter, we will introduce a simulator as an interactive program in Agda. The simulator allows to observe the evolving of processes following external or internal choices.

Chapter 6

A Simulator for CSP-Agda

We have written a simulator in Agda. It turned out to be more complicated than expected, since we needed to convert processes, which are infinite entities, into strings, which are finitary. For instance, if a process is defined recursively as $P = a \xrightarrow{P}$, one cannot extract a finite string from it directly, because P is an infinite object. The solution was to add string components to `Process+` and `Process ∞` . The need to add it to `Process ∞` was unexpected, since `Process+` already seemed to have this information – however, one can only access it if one has a smaller size available. We needed as well to add a conversion of choice sets to labels and restrict result sets to choice sets to make them printable.

The simulator does the following: It will display to the user the selected process, the set of termination choices with their return value (we don't allow the user to follow them, because it will always deadlock), and allows the user to choose an external or internal choice as a string input. If the input is correct, then the program continues with the process, which is obtained by following that transition, otherwise an error message is returned and the program asks again for a choice. The simulator is implemented using a cut down version of the IO library of ooAgda (Abel et al. [2017]), which makes use of the HS-monad. The IO library defines a version `IOConsole` of the IO monad with console commands (`putStrLn s`) for writing a string to console with a return type `Unit`, and `getLine` for getting user input with return type `String`.

The simulator displays the process as a string. Then it computes and displays the set of \checkmark -events and their results, and of external and internal choices together with their labels.

We use the function `choice2Enum` to compute the list of choices, which is defined as follows:

```
choice2Enum : (c : Choice) → List (ChoiceSet c)
```

```

choice2Enum (fin n)           = fin2Option0 n
choice2Enum (c0 ⊔' c1)       = mapL (λ a → inj1 a)
                               (choice2Enum c0) ++
                               mapL (λ a → inj2 a) (choice2Enum c1)
choice2Enum (c0 ×' c1)       = concat (mapL (λ a → (mapL (λ b → (a ,, b))
                               (choice2Enum c1))) (choice2Enum c0))
choice2Enum (namedElements s) = mapL (λ i → ne i) (fin2Option0 (length s))
choice2Enum (Σ' c0 c1)       = concat (mapL (λ a → (mapL (λ b → (a , b))
                               (choice2Enum (c1 a))) (choice2Enum c0))
choice2Enum (subset' E f)     = gfilter (set2MaybeSubset
                               (ChoiceSet E) f) (choice2Enum E)
choice2Enum (list E l)       = mapL lce (fin2Option0 (length l))

```

Here, the function `fin2Option0` is used to create a list of elements of `(Fin n)`. It is defined in Agda as follows:

```

fin2Option0' : (n : ℕ) → List (Fin n)
fin2Option0' zero = []
fin2Option0' (suc n) = last :: mapL embed (fin2Option0' n)

fin2Option0 : (n : ℕ) → List (Fin n)
fin2Option0 n = reverse (fin2Option0' n)

```

The function `choice2Str` creates a string representing a choice, which is defined in Agda as follows:

```

choice2Str : {c : Choice} → ChoiceSet c → String
choice2Str {fin n} m = showN (toN m)
choice2Str {c0 ⊔' c1} (inj1 a) =
  "(inl " ++s (choice2Str {c0} a) ++s ")"
choice2Str {c0 ⊔' c1} (inj2 a) =
  "(inr " ++s (choice2Str {c1} a) ++s ")"
choice2Str {c0 ×' c1} (x ,, x1) =
  "(" ++s (choice2Str {c0} x) ++s
  ",," ++s (choice2Str {c1} x1) ++s ")"
choice2Str {namedElements s} (ne i) = nth s i
choice2Str {Σ' c0 c1} (x1 , x21) =
  (choice2Str {c0} x1) ++s ", "
  ++s (choice2Str {c1 x1} x21)

```

```

choice2Str {subset' E f} (sub a x) = choice2Str {E} a
choice2Str {list E l} (lce i) = choice2Str {E} (nth l i)

```

The simulator will have a parameter *displayProcess*, which determines, whether the user wants to show the full process or hide it. The reason for this is that sometimes the strings representing processes are very big, and the user is interested only in performing the events.

The simulator will operate as follows: It will first display if *displayProcess* is true, the process. It will then check whether the process is the terminated process or it is stuck, i.e. has no external or internal choices.

```

myProgram : ∀ {i} → (displayProcess : Bool) {lu : LUniv}
              (c₀ : Choice)
              → Process ∞ {lu} c₀
              → IOConsole i Unit
forcelO (myProgram {i} true c₀ P) =
              do' (putStrLn (Str P)) λ _ →
              myProgram₀ true c₀ P (proChoicels∅ P)
              (proHasSuccessfullyTerminated P)
myProgram {i} false c₀ P =
              myProgram₀ false c₀ P (proChoicels∅ P)
              (proHasSuccessfullyTerminated P)

```

If it has terminated it displays that the program has terminated and stops. If it is stuck, it displays that the program is stuck and stops. Otherwise, it displays the termination events and external and internal choices available:

```

myProgram₀ : ∀ {i} → (displayProcess : Bool) (c₀ : Choice)
              {lu : LUniv} → Process ∞ {lu} c₀
              → (hasNoInternalOrExternalChoices : Bool)
              → (hasTerminated : Bool)
              → IOConsole i Unit
myProgram₀ displayProcess c₀ P false b =
              do (putStrLn
              ("Termination-Events: " ++ show✓ P)) λ _ →
              do (putStrLn
              ("Events: " ++ showProLab P)) λ _ →
              do (putStrLn ("Choose Event")) λ _ →
              myProgram₁ displayProcess c₀ P
myProgram₀ displayProcess c₀ P true false =

```

```

do (putStrLn "Program got stuck") λ _ →
  return unit
myProgram0 displayProcess c0 P true true =
  do (putStrLn
    "Program has successfully terminated") λ _ →
    return unit

```

The user is now asked to input a string, which is compared to the options "quit" indicating that the user wants to stop, and "showProcess" indicating that the user wants to show the process:

```

myProgram1 : ∀ {i} → (displayProcess : Bool) → (c0 : Choice)
              → {lu : LUniv} → Process ∞ {lu} c0
              → IOConsole i Unit
forcelO (myProgram1 displayProcess c0 P) =
  do' getLine λ s →
    myProgram2 displayProcess c0 P s
    (s ==strb "quit")

```

If the first option is taken the program quits, if the second one is chosen the process is displayed, otherwise the option is compared with the choices available, yielding a *Maybe* applied to the list of external and internal choices.

```

myProgram2 : ∀ {i} → (displayProcess : Bool) (c0 : Choice)
              {lu : LUniv} (P : Process ∞ {lu} c0)
              → String → Bool
              → IOConsole i Unit
myProgram2 displayProcess c0 P s true =
  do (putStrLn "exiting") λ _ →
    return unit
myProgram2 displayProcess c0 P s false =
  myProgram3 displayProcess c0 P s (s ==strb "showProcess")

```

```

myProgram3 : ∀ {i} → (displayProcess : Bool) (c0 : Choice)
              {lu : LUniv} (P : Process ∞ {lu} c0)
              → String → Bool
              → IOConsole i Unit
forcelO (myProgram3 displayProcess c0 P s true) =
  do' (putStrLn (Str P)) λ _ →
    myProgram4 displayProcess c0 P

```

```

      (lookupChoice (proToE P) (proTol P) s)
myProgram3 displayProcess c0 P s false =
      myProgram4 displayProcess c0 P
      (lookupChoice (proToE P) (proTol P) s)

```

If the input was correct, the program follows the external or internal choice chosen by the user. Otherwise the user is asked to enter another choice. Note that \checkmark -events are only displayed but one cannot follow them, because afterwards the system would stop.

```

myProgram4 :  $\forall \{i\} \rightarrow (displayProcess : Bool) (c_0 : Choice) \{lu : LUniv\}$ 
               $(P : Process \infty \{lu\} c_0)$ 
               $\rightarrow Maybe ((ChoiceSet (proToE P)) \uplus (ChoiceSet (proTol P)))$ 
               $\rightarrow IOConsole i Unit$ 
forceIO (myProgram4 displayProcess c0 P nothing) =
      do' (putStrLn "please enter a choice amongst")  $\lambda \_ \rightarrow$ 
      do (putStrLn (showProLab P))  $\lambda \_ \rightarrow$ 
      myProgram1 displayProcess c0 P
forceIO (myProgram4 displayProcess c0 P (just (inj1 ext))) =
      do' (putStrLn
          ("-" ++ showLabel (proToLab P ext) ++ "→" ))
           $\lambda \_ \rightarrow$ 
      myProgram displayProcess c0 (proPToSubPrP P (inj1 ext))
forceIO (myProgram4 displayProcess c0 P (just (inj2 int))) =
      do' (putStrLn "¬→")  $\lambda \_ \rightarrow$ 
      myProgram displayProcess c0 (proPToSubPrP P (inj2 int))

```

Note the occurrence of **force**. In the simultaneous recursive definitions of the programs we need at least one occurrence of **force** in order to guarantee termination.

In CSP-Agda simulator we needed to check if the external and internal choice sets for the process are both empty. In case the set of choices is empty, we obtained that the process got stuck **"Program got stuck"**.

```

proChoicels0 :  $\{ i : Size \} \rightarrow \{ c : Choice \} \rightarrow \{ lu : LUniv \}$ 
                $\rightarrow Process i \{ lu \} c \rightarrow Bool$ 
proChoicels0 (node P) = choicelsEmpty (E P)  $\wedge$  choicelsEmpty (I P)
proChoicels0 (terminate x) = true

```

We define as well a Boolean valued function to check if the process has terminated successfully or not. This function is defined as follows:



```

proHasSuccessfullyTerminated : { i : Size } → { c : Choice } → { lu : LUniv }
                                → Process i { lu } c → Bool
proHasSuccessfullyTerminated (node P) = false
proHasSuccessfullyTerminated (terminate x) = true

```

The CSP-Agda simulator needs to print out the set of labels to the user, who should choose one of them, after which the system proceeds to the next subprocess. We determine the next process as follows:

```

proPToSubPro∞ : ∀ { i } → { c : Choice }
                 → { lu : LUniv }
                 → ( P : Process i { lu } c )
                 → ChoiceSet (proToE P) ⊔ ChoiceSet (proToI P)
                 → Process∞ i { lu } c
proPToSubPro∞ (node P) (inj1 c') = PE P c'
proPToSubPro∞ (node P) (inj2 c') = PI P c'
proPToSubPro∞ (terminate x) (inj1 ())
proPToSubPro∞ (terminate x) (inj2 ())

```

```

proPToSubPrP : ∀ { i } → { j : Size < i } → { c : Choice } → { lu : LUniv }
               → ( P : Process i { lu } c )
               → ChoiceSet (proToE P) ⊔ ChoiceSet (proToI P)
               → Process j c
proPToSubPrP { i } { j } { c } P c' = forcep (proPToSubPro∞ { i } { c } P c')

```

Since we have different kinds of choice, like external, internal and tick event, we need to define a function to show these sorts. Firstly, we define a function, which displays the \checkmark -events, and this is defined as follows:

```

show✓ : { i : Size } → { c : Choice } → { lu : LUniv }
        → Process i { lu } c → String
show✓ (node P) = unlinesWithChosenString
               " "
               (mapL (λ t → (choice2Str t
                               ++s " : "
                               ++s choice2Str (PT P t)))
                     (choice2Enum (T P)))
show✓ (terminate a) = ""

```



In case of external and internal choice, we defined the following function, which creates a string, representing all choices:

```

showProLab : { i : Size } → { c : Choice } → { lu : LUniv }
             → Process i {lu} c → String
showProLab (terminate x) = ""
showProLab {i}{c} (node P) = unlinesWithChosenString
    " "
    ((mapL (λ c' → extChoiceElToName (choice2Str {E P} c')
        ++s " : "
        ++s showLabel (Lab P c'))
    (choice2Enum (E P)))
    ++
    (mapL (λ c → intChoiceElToName (choice2Str c)
        ++s " : "
        ++s "τ")
    (choice2Enum (I P))))

```

If the user entered a displayed event accurately, the system will continue with the next process. Otherwise, the simulator will issue a message to inform the user to enter a choice among the displayed list.

The main function for looking up of choices given by the user is the following function:

```

lookupInEnum : {A : Set} → List (String × A) → String → Maybe A
lookupInEnum [] str = nothing
lookupInEnum ((str' ,, a) :: l) str = lookupInEnumAux a l str
    (str' ==strb str)

lookupInEnumAux : {A : Set} → A → List (String × A) → String → Bool
                → Maybe A
lookupInEnumAux a l s false = lookupInEnum l s
lookupInEnumAux a l s true = just a

```

From this we can define the function `lookupChoice` as follows:

```

combineEnumerations : {E I : Choice} → List (String × ChoiceSet E)
                    → List (String × ChoiceSet I)
                    → List (String × (ChoiceSet E ⊕ ChoiceSet I))
combineEnumerations {E} {I} L L' =
    (mapL (λ {( s ,, c)

```



$$\begin{aligned}
 & \rightarrow (\text{extChoiceElToName } s \text{ ,, } \text{inj}_1 \text{ } c) \} \} L) \\
 & ++ \\
 & (\text{mapL } (\lambda \{ (s \text{ ,, } c) \rightarrow (\text{intChoiceElToName } s \text{ ,, } \text{inj}_2 \text{ } c) \} \} L)') \\
 \text{lookupChoice} & : (E \text{ } I : \text{Choice}) \rightarrow \text{String} \\
 & \rightarrow \text{Maybe } (\text{ChoiceSet } E \uplus \text{ChoiceSet } I) \\
 \text{lookupChoice } E \text{ } I \text{ } s & = \text{lookupInEnum } (\text{combineEnumerations} \\
 & \quad (\text{choice2EnumWithStr } E) \\
 & \quad (\text{choice2EnumWithStr } I)) \text{ } s
 \end{aligned}$$

An example run of the simulator (defined in `IOExampleScreenShotForTyDePaper2.agda`) is as follows:

```

IOExampleScreenShotForTyDePaper2
((b → (a → STOP∞)) □ (((c → STOP∞) ∏ (a → STOP∞)) □ SKIP(STOP)))
Termination-Events: (inr (inr 0)):(inr (inr STOP))
Events: e-(inl 0):b i-(inr (inl 0)):τ i-(inr (inl 1)):τ
Choose Event
i-(inr (inl 0))
((b → (a → STOP∞)) □ ((c → STOP∞) □ SKIP(STOP)))
Termination-Events: (inr (inr 0)):(inr (inr STOP))
Events: e-(inl 0):b e-(inr (inl 0)):c
Choose Event
e-(inl 0)
(fmap (λ STOP → (inl STOP)) (a → STOP∞))
Termination-Events:
Events: e-0:a
Choose Event
e-0
(fmap (λ STOP → (inl STOP)) STOP)
Program got stuck
bashar@bashar-Inspiron-N4030:~/git/papersWithSetzer/PhDThesis/Main/agda$ █

U:**- *shell* All L29 (Shell:run)
the module was successfully compiled.

U:U:%*- *Compilation result* All L1 (AgdaInfo)

```

Internal choice and termination events are labelled by "**i-**" and "**t-**", respectively. Since it is difficult to type in on the terminal inj_1 , inj_2 , we use the traditional names `inl` and `inr` instead. We have in the first step one \checkmark -event (`inr (inr 0)`), one external choice (`inl 0`), and two internal choices (`inr (inl 0)` and `inr (inl 1)`). In this run the user chose one internal choice and then one external choice. We used a more advanced version of `fmap`, which displays a more readable string.



Chapter 7

Trace Semantics for CSP-Agda

In CSP traces of a process are the sequences of actions or labels of external choices, a process can perform. Since processes in CSP are non-deterministic, a process can follow different traces during its execution. The trace semantics of a process is the set of its traces. It captures the observable behaviours of a process. In the next Chapter 8 we will discuss the limitations of trace semantics and how more refined semantics can be used to fix this problem.

Since in CSP-Agda processes are monadic, we need to record, in case after following a trace we obtain a terminated process, the result returned by the process following this trace. So we add a possible element of the result set to the trace. This is a natural extension of the trace semantics of CSP, where a trace is list of labels, which possibly ends with a \checkmark , in case the process has terminated – we just add in the monadic setting to the \checkmark the result returned. We can use for the set of possible elements the set `(Maybe (ChoiceSet c))`. Here the type `(Maybe A)` has elements `(just a)` for $a : A$, denoting defined elements, and an undefined element `nothing`. So `(just a)` denotes that the process has terminated with result a , whereas `nothing` means that it hasn't terminated (or more precisely been determined as terminated).

Taking this together, we obtain that traces are given by a list of labels and an element of `(Maybe (ChoiceSet c))`. We define the set of traces `(Tr l m P)` as a predicate, which determines for a process the lists of labels l and elements $m : \text{Maybe (ChoiceSet c)}$, which form a trace. We define as well traces `(Tr+ l m P)` and `(Tr ∞ l m P)` for processes in `Process+` and `Process ∞` , respectively.

In the trace semantics of CSP a process, which has a termination event has two traces, the empty list, and the list consisting of a \checkmark -event. In order to be consistent with CSP, we will add therefore in case of a termination event or terminated process two traces: the empty list together with possible return value `nothing`, and with possible return value `(just a)` for the return value a .

For an element of $(\text{Process+} \infty c)$ we obtain the following traces:

- The empty trace without termination is a trace of any process, and we denote the proof by **empty**.
- If a process P has external choice x , then from every trace for the result of following this choice, which consisting of a list of labels l and a possible result $tick$, we obtain a trace of P consisting of the result of adding in front of l the label of that external choice, and of the same possible result $tick$. The resulting proof will be denoted by $(\text{extc } l \text{ tick } x \text{ tr})$.
- Internal choices are ignored in traces. Therefore if a process P has an internal choice x , every trace of the result of following this process is a trace of P . The proof is denoted by $(\text{intc } l \text{ tick } x \text{ tr})$
- If a process has a termination event x with return value t , then the empty trace with termination choice $(\text{just } t)$ is a trace of process, having proof $(\text{terc } x)$.

We are going to define the trace semantics. We note that since traces are finite this will be an inductive definition. Later when defining infinite traces we will need a coinductive definition. The definition of traces for the processes of **Process+** is as follows:

mutual

```

data Tr+ {lu : LUniv}{c : Choice} : (l : List (Label lu))
  → (m : Maybe (ChoiceSet c))
  → (P : Process+ ∞ {lu} c) → Set where
empty : {P : Process+ ∞ {lu} c} → Tr+ [] nothing P
extc   : {P : Process+ ∞ {lu} c}
  → (l : List (Label lu))
  → (mc : Maybe (ChoiceSet c))
  → (x : ChoiceSet (E P))
  → (tr : Tr∞ {lu} l mc (PE P x))
  → Tr+ {lu} (Lab P x :: l) mc P
intc   : {P : Process+ ∞ {lu} c}
  → (l : List (Label lu))
  → (mc : Maybe (ChoiceSet c))
  → (x : ChoiceSet (I P))
  → (tr : Tr∞ {lu} l mc (PI P x))
  → Tr+ {lu} l mc P

```

```

terc  : {P : Process+ ∞ {lu} c}
       → (t : ChoiceSet (T P))
       → Tr+ {lu} [] (just (PT P t)) P

```

In case of `Process` we need to consider the termination events:

- The terminated process has two traces, namely the empty list of labels `[]` with termination event `nothing`, and the same list but with termination event `(just x)`, where x is the return value.
- The traces of a non-terminated process are the traces of the corresponding element of `Process+`.

We obtain the following definition of the traces of `Process`:

```

data Tr {lu : LUniv}{c : Choice} : (l : List (Label lu))
    (m : Maybe (ChoiceSet c))
    (P : Process ∞ {lu} c) → Set where
ter  : (x : ChoiceSet c) → Tr {lu} [] (just x) (terminate x)
empty : (x : ChoiceSet c) → Tr {lu} [] nothing (terminate x)
tnode : {l : List (Label lu)}
       → {x : Maybe (ChoiceSet c)}
       → {P : Process+ ∞ {lu} c}
       → (tr : Tr+ {lu} {c} l x P)
       → Tr {lu} l x (node P)

```

Finally the traces for `Process∞` are just the traces of the underlying `Process`:

```

Tr∞ : {lu : LUniv}{c : Choice} (l : List (Label lu))
    (tick : Maybe (ChoiceSet c))
    (P : Process∞ ∞ {lu} c) → Set
Tr∞ {c} l tick P = Tr l tick (forceP P)

```

In CSP, a process P refines a process Q , written $(P \sqsubseteq Q)$ if and only if any observable behaviour of Q is an observable behaviour of P , i.e. if $traces(Q) \subseteq traces(P)$:

```

_⊆_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
    (Q : Process ∞ {lu} c) → Set
_⊆_ {lu} {c} P Q = (l : List (Label lu))
    → (m : Maybe (ChoiceSet c))

```

$$\rightarrow (tr : \text{Tr } \{lu\} \ l \ m \ Q) \rightarrow \text{Tr } \{lu\} \ l \ m \ P$$

Two processes P , Q are equal with respect to trace semantics, written $P \equiv Q$, if they refine each other, i.e. if $traces(P) = traces(Q)$:

$$\begin{aligned} _ \equiv _ &: \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process } \infty \ \{lu\} \ c_0) \rightarrow \text{Set} \\ P \equiv Q &= P \sqsubseteq Q \times Q \sqsubseteq P \end{aligned}$$

7.1 Proof of the Algebraic Laws

In CSP there are many algebraic laws for individual operators, and also concerning the relationships between different operators. Numerous laws are concerned with general algebraic properties such as commutativity and associativity of operators: these properties allow a process to be composed in any order, the identification of zeros and units for specific operators, idempotence, etc. Other laws allow process descriptions to be simplified. Many laws are concerned with the relationships between different operators, for example the expansion of a parallel into a prefix choice process. We will present examples of how to prove algebraic laws of CSP in Agda using this semantics.

7.2 Proof of the Laws of Refinement

The refinement relation is reflexive, anti-symmetric and transitive, i.e. fulfils the following laws:

$$\begin{aligned} P &\sqsubseteq P \\ P_0 &\sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 = P_1 \\ P_0 &\sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2 \end{aligned}$$

For the above relations, the definition is given by stating that if the second process fulfils a certain property (e.g. that tr is a trace) the first process fulfils it as well. They are equivalent, if refinement goes in both directions. This implies immediately reflexivity, antisymmetry, and transitivity. Note that antisymmetry is trivial, since the conclusion is defined as the conjunction of the two antecedents.

Theorem 7.2.1 (Agda Theorem)

$$\text{refl} \sqsubseteq : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process } \infty \ \{lu\} \ c) \rightarrow P \sqsubseteq P$$

$\text{trans} \sqsubseteq : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$
 $(Q : \text{Process} \infty \{lu\} c)$
 $(R : \text{Process} \infty \{lu\} c) \rightarrow P \sqsubseteq Q \rightarrow Q \sqsubseteq R \rightarrow P \sqsubseteq R$
 $\text{antiSym} \sqsubseteq : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} \infty \{lu\} c_0)$
 $\rightarrow P \sqsubseteq Q \rightarrow Q \sqsubseteq P \rightarrow P \equiv Q$

Proof:

$\text{refl} \sqsubseteq \quad P \ l \ m \ x \quad = \ x$
 $\text{trans} \sqsubseteq \quad P \ Q \ R \ PQ \ QR \ l \ m \ tr = PQ \ l \ m \ (QR \ l \ m \ tr)$
 $\text{antiSym} \sqsubseteq \quad P \ Q \ PQ \ QP \quad = \ PQ \ , \ QP$

7.3 Proof of the Monadic Laws

We defined processes in a monadic way, and will in this section prove the monad laws for processes.

In functional programming, a monad is given by a functor \mathbf{M} together with morphisms $\gg= : \mathbf{M} \ A \rightarrow (A \rightarrow \mathbf{M} \ B) \rightarrow \mathbf{M} \ B$ and $\text{return} : A \rightarrow \mathbf{M} \ A$ such that the following laws hold:

$$\begin{aligned}
 \text{return } a \gg= f &= f \ a \\
 p \gg= \text{return} &= p \\
 (p \gg= f) \gg= g &= p \gg= (\lambda x. f \ x \gg= g)
 \end{aligned}$$

For each monadic law we have to prove 2 directions, (“ \sqsubseteq ” and “ \sqsupseteq ”). Furthermore the laws need to be shown for Process+ , Process and $\text{Process}\infty$. We will present only one direction and one version of the processes for each law. Since proofs of $_ \equiv _$ just follow from the left to right and right to left refinement, we will present this proof only for the first monadic law.

The proof of the first monadic law is trivial since $(\text{terminate } a \gg= P)$ is definitionally to P ; one could as well have used reflexivity of \sqsubseteq and \equiv :

Theorem 7.3.1 (Agda Theorem)

$\equiv \text{monadicLaw}_1 : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1)$
 $\rightarrow (P \ a) \equiv (\text{terminate } a \gg= P)$

Proof:

$$\begin{aligned} \text{monadicLaw}_1 &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\quad (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\ &\quad \rightarrow (\text{terminate } a \gg= P) \sqsubseteq P \ a \\ \text{monadicLaw}_1 \ a \ P \ l \ m \ q &= q \end{aligned}$$

$$\begin{aligned} \text{monadicLaw}_1 R &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\quad (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\ &\quad \rightarrow P \ a \sqsubseteq (\text{terminate } a \gg= P) \\ \text{monadicLaw}_1 R \ \{c_0\} \ \{c_1\} \ a \ P \ l \ m \ q &= q \end{aligned}$$

$$\begin{aligned} &\equiv \text{monadicLaw}_1 : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\quad (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\ &\quad \rightarrow (P \ a) \equiv (\text{terminate } a \gg= P) \\ &\equiv \text{monadicLaw}_1 \ \{c_0\} \ \{c_1\} \ a \ P = (\text{monadicLaw}_1 \ a \ P) , (\text{monadicLaw}_1 R \ a \ P) \end{aligned}$$

In case of the second monadic law the proof is by induction over the proofs of traces for $(P \gg= + \text{terminate})$, which immediately turn into traces of P .

Theorem 7.3.2 (Agda Theorem)

$$\begin{aligned} \equiv \text{monadicLaw}_2 &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\ &\quad (P : \text{Process } \infty \{lu\} \ c_0) \\ &\quad \rightarrow P \equiv (P \gg= \text{terminate}) \end{aligned}$$

Proof:

$$\begin{aligned} \text{monadicLaw}_{2+} &: \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (P : \text{Process} + \infty \{lu\} \ c_0) \\ &\quad \rightarrow (P \gg= + \text{terminate}) \sqsubseteq + P \\ \text{monadicLaw}_{2+} \ P \ .[] \ .\text{nothing empty} &= \text{empty} \\ \text{monadicLaw}_{2+} \ P \ .(\text{Lab } P \ x :: l) \ m \ (\text{extc } l \ .m \ x \ x_1) &= \\ &\quad \text{extc } l \ m \ x \\ &\quad (\text{monadicLaw}_2 \infty (\text{PE } P \ x) \ l \ m \ x_1) \\ \text{monadicLaw}_{2+} \ P \ l \ m \ (\text{intc } l \ .m \ x \ x_1) &= \\ &\quad \text{intc } l \ m \ (\text{inj}_1 \ x) \\ &\quad (\text{monadicLaw}_2 \infty (\text{PI } P \ x) \ l \ m \ x_1) \\ \text{monadicLaw}_{2+} \ \{lu\} \ \{c_0\} \ P \ .[] \ .(\text{just } (\text{PT } P \ x)) \ (\text{terc } x) &= \end{aligned}$$

$$\text{intc } [] \text{ (just (PT } P \text{ } x)) \text{ (inj}_2 \text{ } x) \\ (\text{lemTrTerminateBind } c_0 \text{ } P \text{ } x)$$

$$\begin{aligned} &\equiv \text{monadicLaw}_2 : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (P : \text{Process} \infty \{lu\} \ c_0) \\ &\quad \rightarrow P \equiv (P \gg= \text{terminate}) \\ &\equiv \text{monadicLaw}_2 \ \{lu\} \ \{c_0\} \ \{c_1\} \ P = (\text{monadicLaw}_2 \text{R } P) \ , (\text{monadicLaw}_2 \ P) \end{aligned}$$

The proof of the third monadic law is by induction over the proofs of traces for $(P \gg=+ (Q \gg=+ R))$. In most cases the proof of traces carry over after applying the induction hypothesis. One special case if the first process P has a termination event, which results in an internal choice to $Q \ x \gg= R$ on both sides. In this case the traces are essentially the same, but only after applying **forget**. We use here an operation

$$\begin{aligned} \text{monadPT+} : &\{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process+} \infty \{lu\} \ c_0) \\ &\quad (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} \ c_1) \\ &\rightarrow (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} \ c_2) \\ &\rightarrow (y : \text{ChoiceSet } (\text{T } P)) \\ &\rightarrow (l : \text{List } (\text{Label } lu)) \\ &\rightarrow (m : \text{Maybe } (\text{ChoiceSet } c_2)) \\ &\rightarrow (x : \text{Tr} \infty \ l \ m \ (\text{PI } (P \gg=+ (\lambda x \rightarrow Q \ x \gg= R)) \text{ (inj}_2 \ y)))) \\ &\rightarrow \text{Tr} \infty \ l \ m \ (\text{PI } (P \gg=+ Q) \text{ (inj}_2 \ y) \gg= \infty R) \\ \text{monadPT+ } &\{lu\} \ \{c_0\} \ \{c_1\} \ \{c_2\} \ P \ Q \ R \ y \ l \ m \ tr = tr \end{aligned}$$

which is modulo an application of **forget** equal to tr . There are no immediate termination events, and therefore no proofs of traces of the form $(\text{terc } x)$. We use **efq** (ex falso quodlibet), which constructs from an element of the empty set an element of any set, for dealing with this case. The resulting theorem and proof is as follows:

Theorem 7.3.3 (Agda Theorem)

$$\begin{aligned} \equiv \text{monadicLaw}_3 : &\{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} \\ &\quad (P : \text{Process} \infty \{lu\} \ c_0) \\ &\quad (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} \ c_1) \\ &\quad (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} \ c_2) \\ &\rightarrow ((P \gg= Q) \gg= R) \equiv (P \gg= (\lambda x \rightarrow Q \ x \gg= R)) \end{aligned}$$

Proof:

$$\text{monadicLaw}_{3+} : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process+} \infty \{lu\} \ c_0)$$

$$\begin{aligned}
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) \\
& \rightarrow ((P \gg=+ Q) \gg=+ R) \sqsubseteq + \\
& \quad (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\
\text{monadicLaw}_{3+} \quad & P Q R . [] . \text{nothing empty} = \text{empty} \\
\text{monadicLaw}_{3+} \quad & P Q R . (\text{Lab } P x :: l) m (\text{extc } l . m x x_1) = \\
& \quad \text{extc } l m x \\
& \quad (\text{monadicLaw}_\infty P Q R l x m x_1) \\
\text{monadicLaw}_{3+} \quad & P Q R l m (\text{intc } . l . m (\text{inj}_1 x) x_1) = \\
& \quad \text{intc } l m (\text{inj}_1 (\text{inj}_1 x)) \\
& \quad (\text{monadicLaw}_{3\infty} (\text{PI } P x) Q R l m x_1) \\
\text{monadicLaw}_{3+} \quad & P Q R l m (\text{intc } . l . m (\text{inj}_2 y) x_1) = \\
& \quad \text{intc } l m (\text{inj}_1 (\text{inj}_2 y)) \\
& \quad (\text{monadPT} + P Q R y l m x_1) \\
\text{monadicLaw}_{3+} \quad & P Q R . [] . (\text{just } (\text{PT} \\
& \quad (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) x)) (\text{terc } x) = \text{efq } x \\
\\
\equiv \text{monadicLaw}_3 : & \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process } \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) \\
& \rightarrow ((P \gg= Q) \gg= R) \equiv (P \gg= (\lambda x \rightarrow Q x \gg= R)) \\
\equiv \text{monadicLaw}_3 \quad & \{lu\} \{c_0\} \{c_1\} \{c_2\} P Q R = (\text{monadicLaw}_3 P Q R) , \\
& \quad (\text{monadicLaw}_{3R} P Q R)
\end{aligned}$$

We prove now a law which expresses $\text{STOP} \circ P = \text{STOP}$: if we compose the STOP process with any process we get the STOP process back. The proof of this law is as follows:

$$\begin{aligned}
\text{stopSeq} : & \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\
& (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_0) \\
& \rightarrow (\text{STOP } c_0 \gg= P) \sqsubseteq \text{STOP } c_0 \\
\text{stopSeq } a P . [] . \text{nothing } (\text{tnode empty}) &= \text{tnode empty} \\
\text{stopSeq } a P . (\text{efq } _ :: l) m (\text{tnode } (\text{extc } l . m () x_1)) &= \\
\text{stopSeq } a P l m (\text{tnode } (\text{intc } . l . m () x_1)) &= \\
\text{stopSeq } a P . [] . (\text{just } (\text{efq } _)) (\text{tnode } (\text{terc } ())) &=
\end{aligned}$$

Theorem 7.3.4 (Agda Theorem)

$$\equiv \text{stopSeq} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$$

$$\begin{aligned} & (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_0) \\ & \rightarrow \text{STOP } c_0 \{lu\} \equiv (\text{STOP } c_0 \{lu\} \gg= P) \end{aligned}$$

Proof:

$$\begin{aligned} & \equiv \text{stopSeq} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ & \quad (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_0) \\ & \quad \rightarrow \text{STOP } c_0 \{lu\} \equiv (\text{STOP } c_0 \{lu\} \gg= P) \\ & \equiv \text{stopSeq } a P = (\text{stopSeqr } a P) , (\text{stopSeq } a P) \end{aligned}$$

7.4 Proof of Commutativity of the Interleaving Operator

The interleaving combination $(P \parallel Q)$ executes each component completely independent of the other, until termination. Traces of the interleaving combination $P \parallel Q$ will, therefore, appear as interleavings of traces of the two component, and therefore it is easy to see that $(P \parallel Q)$ and $(Q \parallel P)$ are trace equivalent.

However, because of the monadic setting, for most algebraic laws the return types of the left and right hand side of an equation are different. Assume the return types of P and Q are c_0 and c_1 , respectively. Then for instance the return type of $(P \parallel Q)$ is $(c_0 \times' c_1)$ whereas the return type of $(Q \parallel P)$ is $(c_1 \times' c_0)$.

Therefore the algebraic laws hold only modulo applying an adjustment of the return types using the operation `fmap`, which applies a function to the return types.

Once we have taken this into account, a proof of commutativity of $_{\parallel}$ is obtained by exchanging the external/internal/termination choices, which means swapping `inj1` and `inj2`. Here `inj1` refers to choices in the first and `inj2` to choices in the second process.

We give here the main case referring to `Process+` (`swap ×` swaps the two sides of a product):

$$\begin{aligned} & S \parallel + : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c_0) \\ & \quad (Q : \text{Process+ } \infty \{lu\} c_1) \\ & \quad \rightarrow (P \parallel + Q) \sqsubseteq + (\text{fmap+ swap } \times (Q \parallel + P)) \\ & S \parallel + P Q . [] . \text{nothing empty} = \text{empty} \\ & S \parallel + P Q . (\text{Lab } Q x :: l) m (\text{extc } l . m (\text{inj}_1 x) q) = \\ & \quad \text{extc } l m (\text{inj}_2 x) (S \parallel + P (\text{PE } Q x) l m q) \end{aligned}$$

$$\begin{aligned}
S|||+ P Q .(\text{Lab } P x :: l) m (\text{extc } l .m (\text{inj}_2 x) q) &= \\
&\text{extc } l m (\text{inj}_1 x)(S|||\infty+ (\text{PE } P x) Q l m q) \\
S|||+ P Q l m (\text{intc } .l .m (\text{inj}_1 x) q) &= \\
&\text{intc } l m (\text{inj}_2 x)(S|||+\infty P (\text{PI } Q x) l m q) \\
S|||+ P Q l m (\text{intc } .l .m (\text{inj}_2 x) q) &= \\
&\text{intc } l m (\text{inj}_1 x)(S|||\infty+ (\text{PI } P x) Q l m q) \\
S|||+ P Q .[] .(\text{just } (\text{PT } P x ,, \text{PT } Q y)) (\text{terc } (y ,, x)) &= \\
&\text{terc } (x ,, y)
\end{aligned}$$

Theorem 7.4.1 (Agda Theorem)

$$\begin{aligned}
\equiv S|||+ : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} \\
(P : \text{Process}+ \infty \{lu\} c_0) \\
(Q : \text{Process}+ \infty \{lu\} c_1) \\
\rightarrow (P |||++ Q) \equiv + (\text{fmap}+ \text{swap} \times (Q |||++ P))
\end{aligned}$$

Proof:

$$\begin{aligned}
\equiv S|||+ : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c_0) \\
(Q : \text{Process}+ \infty \{lu\} c_1) \\
\rightarrow (P |||++ Q) \equiv + (\text{fmap}+ \text{swap} \times (Q |||++ P)) \\
\equiv S|||+ P Q = (S|||+ P Q) , (S|||+R P Q)
\end{aligned}$$

7.5 Proof of Commutativity of the Parallel Operator

Most cases in the proof of the commutativity of $[-]|||+[-]$ are similar to the proof of commutativity $|||$ – one swaps inj_1 and inj_2 and uses induction. The only more difficult case is when we have two processes synchronising, resulting in both processes following choices having the same labels. This case uses a proof that the two choices for the two processes result have the same label and that both labels are in the synchronised sets. We obtain in this case from a proof that we have a trace a proof of the Boolean conjunction:

$$\text{Lab } Q x == | \text{Lab } P x_1 \wedge B (\text{Lab } Q x) \wedge A (\text{Lab } P x_1)$$

which we need to transform into a proof of the Boolean conjunction

$$\text{Lab } P x_1 == | \text{Lab } Q x \wedge A (\text{Lab } P x_1) \wedge B (\text{Lab } Q x)$$

We will make use of functions, which introduce and eliminate proofs of Boolean conjunctions, i.e.

```

 $\wedge\text{BoolIntro} \quad : \quad (a \ b : \text{Bool}) \rightarrow \mathsf{T} \ a \rightarrow \mathsf{T} \ b \rightarrow \mathsf{T} \ (a \wedge b)$ 
 $\text{lemmaBool} \quad : \quad (a \ b : \text{Bool}) \rightarrow \mathsf{T} \ (a \wedge b) \rightarrow \mathsf{T} \ a$ 
 $\text{lemmaBoolR} \quad : \quad (a \ b : \text{Bool}) \rightarrow \mathsf{T} \ (a \wedge b) \rightarrow \mathsf{T} \ b$ 

```

Furthermore, we make use of a proof `sym` of symmetry of the Boolean equality `_==_` on labels, and the transfer lemma

```

 $\text{transf} : (Q : \text{Label} \rightarrow \text{Set}) \rightarrow (l \ l' : \text{Label}) \rightarrow \mathsf{T} \ (l == l') \rightarrow Q \ l \rightarrow Q \ l'$ 

```

`lemmaBool` and `lemmaBoolR` are defined as follows:

```

 $\text{lemmaBool} : (a \ b : \text{Bool}) \rightarrow \mathsf{T}' \ (a \wedge b) \rightarrow \mathsf{T}' \ a$ 
 $\text{lemmaBool} \ \text{false} \ b \ ()$ 
 $\text{lemmaBool} \ \text{true} \ \text{false} \ ()$ 
 $\text{lemmaBool} \ \text{true} \ \text{true} \ \text{tt} = \text{tt}$ 

 $\text{lemmaBoolR} : (a \ b : \text{Bool}) \rightarrow \mathsf{T}' \ (a \wedge b) \rightarrow \mathsf{T}' \ b$ 
 $\text{lemmaBoolR} \ \text{false} \ \text{false} \ ()$ 
 $\text{lemmaBoolR} \ \text{true} \ \text{false} \ ()$ 
 $\text{lemmaBoolR} \ \text{false} \ \text{true} \ ()$ 
 $\text{lemmaBoolR} \ \text{true} \ \text{true} \ \text{tt} = \text{tt}$ 

```

We define as well a function which eliminates a conjunction of 3 elements:

```

 $\text{lemmaBool'aux} : (a \ b \ c : \text{Bool})$ 
 $\quad \rightarrow \mathsf{T}' \ (a \wedge b \wedge c) \rightarrow \mathsf{T}' \ c$ 
 $\text{lemmaBool'aux} \ a \ b \ c \ p = \text{let}$ 
 $\quad q : \mathsf{T}' \ (b \wedge c)$ 
 $\quad q = \text{lemmaBoolR} \ a \ (b \wedge c) \ p$ 
 $\quad \text{in lemmaBoolR} \ b \ c \ q$ 

```

Then we apply the proof of symmetry to the equality proof and recombine them. Finally, we need to carry out a transfer to replace the first label (`Lab P x1`) in the trace by (`Lab Q x`), which are known to be equal. The resulting proof is as follows:

```

 $S[[]]_+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (P : \text{Process} \rightarrow \infty \ \{lu\} \ c_0)$ 
 $\quad (A \ B : \text{Label} \ lu \rightarrow \text{Bool})$ 

```

$$\begin{aligned}
& (Q : \text{Process} + \infty \{lu\} c_1) \\
& \rightarrow (P [A] ||+ [B] Q) \sqsubseteq + \text{fmap} + \text{swap} \times ((Q [B] ||+ [A] P)) \\
\\
& S[||]+ P A B Q . [] . \text{nothing empty} = \text{empty} \\
& S[||]+ P A B Q . (\text{Lab } Q a :: l) m (\text{extc } l m (\text{inj}_1 (\text{inj}_1 (\text{sub } a x))) x_1) = \\
& \quad \text{extc } l m (\text{inj}_1 (\text{inj}_2 (\text{sub } a x))) (S[||]+ \infty P A B (\text{PE } Q a) l m x_1) \\
& S[||]+ P A B Q . (\text{Lab } P a :: l) m (\text{extc } l m (\text{inj}_1 (\text{inj}_2 (\text{sub } a x))) x_1) = \\
& \quad \text{extc } l m (\text{inj}_1 (\text{inj}_1 (\text{sub } a x))) (S[||]+ \infty + (\text{PE } P a) A B Q l m x_1) \\
& S[||]+ \{lu\} P A B Q . (\text{Lab } Q x :: l) m (\text{extc } l m (\text{inj}_2 (\text{sub } (x ,, x_1) x_2)) x_3) = \\
& \quad \text{let} \\
& \quad lxlx_1 : T' (\text{Lab } Q x ==| \text{Lab } P x_1) \\
& \quad lxlx_1 = \text{lemmaBool} (\text{Lab } Q x ==| \text{Lab } P x_1) \\
& \quad \quad (B (\text{Lab } Q x) \wedge A (\text{Lab } P x_1)) x_2 \\
\\
& BQx : T' (B (\text{Lab } Q x)) \\
& BQx = \text{lemmaBool} (B (\text{Lab } Q x)) (A (\text{Lab } P x_1)) \\
& \quad (\text{lemmaBoolR} ((\text{Lab } Q x ==| \text{Lab } P x_1)) \\
& \quad \quad (B (\text{Lab } Q x) \wedge A (\text{Lab } P x_1)) x_2) \\
\\
& APx_1 : T' (A (\text{Lab } P x_1)) \\
& APx_1 = \text{lemmaBool}' ((\text{Lab } Q x ==| \text{Lab } P x_1)) \\
& \quad (B (\text{Lab } Q x)) (A (\text{Lab } P x_1)) x_2 \\
\\
& lxlx : T' (\text{Lab } P x_1 ==| \text{Lab } Q x) \\
& lxlx = \text{sym} ==| \{lu\} \{\text{Lab } Q x\} \{\text{Lab } P x_1\} lxlx_1 \\
\\
& x_2' : T' ((\text{Lab } P x_1 ==| \text{Lab } Q x) \\
& \quad \wedge A (\text{Lab } P x_1) \wedge B (\text{Lab } Q x)) \\
& x_2' = \text{lemmaBool}'' (\text{Lab } P x_1 ==| \text{Lab } Q x) \\
& \quad (A (\text{Lab } P x_1)) (B (\text{Lab } Q x)) \\
& \quad lxlx APx_1 BQx \\
\\
& auxproof : Tr+ (\text{Lab } P x_1 :: l) m \\
& \quad (P [A] ||+ [B] Q) \\
& auxproof = \text{extc } l m (\text{inj}_2 (\text{sub } (x_1 ,, x) x_2')) \\
& \quad (S[||]+ \infty \infty (\text{PE } P x_1) A B (\text{PE } Q x) l m x_3) \\
\\
& auxproof' : Tr+ (\text{Lab } Q x :: l) m (P [A] ||+ [B] Q)
\end{aligned}$$

$$\begin{aligned}
auxproof' = & \text{transfLu } \{lu\} (\lambda l' \rightarrow \text{Tr+ } (l' :: l) \\
& m (P [A] || + [B] Q)) \{ \text{Lab } P x_1 \} \\
& \{ \text{Lab } Q x \} \text{ } lx_1 \text{ } lx \text{ } auxproof \\
& \text{in } auxproof'
\end{aligned}$$

$$\begin{aligned}
S[[]]+ P A B Q l m (\text{intc } .l .m (\text{inj}_1 x) x_1) = & \\
& \text{intc } l m (\text{inj}_2 x) (S[[]]+ \infty P A B (\text{PI } Q x) l m x_1) \\
S[[]]+ P A B Q l m (\text{intc } .l .m (\text{inj}_2 y) x_1) = & \\
& \text{intc } l m (\text{inj}_1 y) (S[[]]+ \infty (\text{PI } P y) A B Q l m x_1) \\
S[[]]+ P A B Q .[] .(\text{just } (\text{PT } P x_1 ,, \text{PT } Q x)) (\text{terc } (x ,, x_1)) = & \text{terc } (x_1 ,, x)
\end{aligned}$$

Theorem 7.5.1 (Agda Theorem)

$$\begin{aligned}
\equiv S[[]]+ : & \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} \\
& (P : \text{Process+ } \infty \{lu\} c_0) \\
& (A B : \text{Label } lu \rightarrow \text{Bool}) \\
& (Q : \text{Process+ } \infty \{lu\} c_1) \\
& \rightarrow (P [A] || + [B] Q) \equiv + (\text{fmap+ swap} \times ((Q [B] || + [A] P)))
\end{aligned}$$

Proof:

$$\begin{aligned}
\equiv S[[]]+ : & \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c_0) \\
& (A B : \text{Label } lu \rightarrow \text{Bool}) \\
& (Q : \text{Process+ } \infty \{lu\} c_1) \\
& \rightarrow (P [A] || + [B] Q) \equiv + (\text{fmap+ swap} \times ((Q [B] || + [A] P))) \\
\equiv S[[]]+ P A B Q = & (S[[]]+ P A B Q) , (S[[]]+ R P A B Q)
\end{aligned}$$

7.6 Proof of Commutativity of the External Choice Operator

The traces of the external choice $(P \sqcup Q)$ of processes are the external choice of the traces of the two components. Therefore it is easy to see that $(P \sqcup Q)$ and $(Q \sqcup P)$ are trace equivalent.

However, because of the monadic setting, the return types of the left and right-hand side of the equation are different. Assume the return types of P and Q are c_0 and c_1 , respectively. Then, the return type of $(P \sqcup Q)$ is $(c_0 \uplus' c_1)$, whereas the return type of $(Q \sqcup P)$ is $(c_1 \uplus' c_0)$. Therefore the

algebraic laws hold only modulo applying an adjustment of the return types using the operation `fmap`, which applies a function to the return values. Such adjustments need to be made to most other algebraic laws.

Once we have taken this into account, a proof of commutativity of `_□_` is obtained by exchanging the external/internal/termination choices of the left and right process. Since `inj1` refers to choices in the first and `inj2` to choices in the second process, it is obtained by swapping `inj1` and `inj2`. We give here the main case referring to `Process+` (`swap⊕` is the function swapping the two sides of a disjoint union). This proof is by induction in the two processes, which in Agda turns into a recursive proof:

```

S□+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
  → (P □++ Q) □+ (fmap+ swap⊕ (Q □++ P))
S□+ P Q .[] .nothing empty = empty
S□+ P Q .(Lab Q x :: l) m (extc l .m (inj1 x) x1) = extc l m (inj2 x)
      (lemFmap∞ inj1 swap⊕ (PE Q x) l m x1)
S□+ P Q .(Lab P y :: l) m (extc l .m (inj2 y) x1) = extc l m (inj1 y)
      (lemFmap∞ inj2 swap⊕ (PE P y) l m x1)
S□+ P Q l m (intc .l .m (inj1 x) x1) = intc l m (inj2 x)
      (S□+∞ P (PI Q x) l m x1)
S□+ P Q l m (intc .l .m (inj2 y) x1) = intc l m (inj1 y)
      (S□+∞ (PI P y) Q l m x1)
S□+ P Q .[] .(just (inj2 (PT Q x))) (terc (inj1 x)) = terc (inj2 x)
S□+ P Q .[] .(just (inj1 (PT P y))) (terc (inj2 y)) = terc (inj1 y)

```

Theorem 7.6.1 (Agda Theorem)

```

≡□+ : {lu : LUniv}{c0 c1 : Choice}
      (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
  → (P □++ Q) ≡+ (fmap+ swap⊕ (Q □++ P))

```

Proof:

```

≡□+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
  → (P □++ Q) ≡+ (fmap+ swap⊕ (Q □++ P))
≡□+ P Q = S□+ P Q , S□+R P Q

```

7.7 Proof of Associativity of the Interleaving Operator

Traces of the interleaving combination $((P \parallel Q) \parallel Z)$ will appear as interleavings of traces of the three component, and therefore it is easy to see that $((P \parallel Q) \parallel Z)$ and $(P \parallel (Q \parallel Z))$ are trace equivalent.

However, as before, the return types of the left and right hand side of an equation are different. Assume the return types of P , Q and Z are c_0 , c_1 and c_2 respectively. Then for instance the return type of $((P \parallel Q) \parallel Z)$ is $((c_0 \times' c_1) \times' c_2)$ whereas the return type of $(P \parallel (Q \parallel Z))$ is $(c_0 \times' (c_1 \times' c_2))$.

Therefore the algebraic laws hold only modulo applying an adjustment of the return types using the operation `fmap`, which applies a function to the return types.

Once we have taken this into account, a proof of associativity of $_||_$ is obtained by exchanging the bracket between external/internal/termination choices, which means exchange the bracket between `inj1 inj1 x`, `inj1 inj2 x` and `inj2 x`. Here `inj1 inj1 x` refers to choices in the first, `inj1 inj2 x` to choices in the second process and `inj2 x` to choices in the third process.

We give here the main case referring to `Process+` (`swap` swaps the bracket sides of a product):

$$\begin{aligned}
& \text{Ass}|||+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process+} \ \infty \ \{lu\} \ c_0) \\
& \quad (Q : \text{Process+} \ \infty \ \{lu\} \ c_1) \\
& \quad (Z : \text{Process+} \ \infty \ \{lu\} \ c_2) \\
& \rightarrow ((P |||++ Q) |||++ Z) \sqsubseteq+ \text{fmap+ Ass} \times (P |||++ (Q |||++ Z)) \\
& \text{Ass}|||+ P \ Q \ Z \ .[] \ .\text{nothing empty} = \text{empty} \\
& \text{Ass}|||+ P \ Q \ Z \ .(\text{Lab } P \ x :: l) \ m \ (\text{extc } l \ .m \ (\text{inj}_1 \ x) \ x_1) \\
& \quad = \text{extc } l \ m \ (\text{inj}_1 \ (\text{inj}_1 \ x)) \ (\text{Ass}|||\infty++ (\text{PE } P \ x) \ Q \ Z \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ .(\text{Lab } Q \ x :: l) \ m \ (\text{extc } l \ .m \ (\text{inj}_2 \ (\text{inj}_1 \ x)) \ x_1) \\
& \quad = \text{extc } l \ m \ (\text{inj}_1 \ (\text{inj}_2 \ x)) \ (\text{Ass}|||+\infty++ P \ (\text{PE } Q \ x) \ Z \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ .(\text{Lab } Z \ y :: l) \ m \ (\text{extc } l \ .m \ (\text{inj}_2 \ (\text{inj}_2 \ y)) \ x_1) \\
& \quad = \text{extc } l \ m \ (\text{inj}_2 \ y) \ (\text{Ass}|||+\infty P \ Q \ (\text{PE } Z \ y) \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ l \ m \ (\text{intc } l \ .m \ (\text{inj}_1 \ x) \ x_1) \\
& \quad = \text{intc } l \ m \ (\text{inj}_1 \ (\text{inj}_1 \ x)) \ (\text{Ass}|||\infty++ (\text{PI } P \ x) \ Q \ Z \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ l \ m \ (\text{intc } l \ .m \ (\text{inj}_2 \ (\text{inj}_1 \ x)) \ x_1) \\
& \quad = \text{intc } l \ m \ (\text{inj}_1 \ (\text{inj}_2 \ x)) \ (\text{Ass}|||+\infty++ P \ (\text{PI } Q \ x) \ Z \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ l \ m \ (\text{intc } l \ .m \ (\text{inj}_2 \ (\text{inj}_2 \ y)) \ x_1) \\
& \quad = \text{intc } l \ m \ (\text{inj}_2 \ y) \ (\text{Ass}|||+\infty P \ Q \ (\text{PI } Z \ y) \ l \ m \ x_1) \\
& \text{Ass}|||+ P \ Q \ Z \ .[] \ .(\text{just } ((\text{PT } P \ x ,, \text{PT } Q \ x_1) ,, \text{PT } Z \ x_2)) \\
& \quad \quad (\text{terc } (x ,, (x_1 ,, x_2))) = \text{terc } ((x ,, x_1) ,, x_2)
\end{aligned}$$

Theorem 7.7.1 (Agda Theorem)

$$\begin{aligned}
& \equiv |||+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} \\
& \quad (P : \text{Process+} \ \infty \ \{lu\} \ c_0) \\
& \quad (Q : \text{Process+} \ \infty \ \{lu\} \ c_1) \\
& \quad (Z : \text{Process+} \ \infty \ \{lu\} \ c_2) \\
& \rightarrow ((P \ |||++ \ Q) \ |||++ \ Z) \equiv+ \ (\text{fmap+} \ \text{Ass}\times \ (P \ |||++ \ (Q \ |||++ \ Z)))
\end{aligned}$$

Proof:

$$\equiv |||+ \ P \ Q \ Z = \text{Ass}|||+ \ P \ Q \ Z, \text{Ass}|||+R \ P \ Q \ Z$$

7.8 Proof of Associativity of the External Choice Operator

The traces of the external choice $((P \sqcup Q) \sqcup Z)$ of processes are the external choice of the traces of the three components. Therefore it is easy to see that $((P \sqcup Q) \sqcup Z)$ and $(P \sqcup (Q \sqcup Z))$ are trace equivalent.

As before the return types need to be adjusted. Assume the return types of P , Q and Z are c_0 , c_1 and c_2 , respectively. Then, the return type of $((P \sqcup Q) \sqcup Z)$ is $((c_0 \uplus' c_1) \uplus' c_2)$, whereas the return type of $(P \sqcup (Q \sqcup Z))$ is $(c_0 \uplus' (c_1 \uplus' c_2))$. Therefore the algebraic laws hold only modulo applying an adjustment of the return types using the operation `fmap`, which applies a function to the return values.

Once we have taken this into account, a proof of associativity of `_□_` is obtained by exchanging the bracket between external/internal/termination choices, which means exchange the bracket between `inj1 inj1 x`, `inj1 inj2 x` and `inj2 x`. Here `inj1 inj1 x` refers to choices in the first, `inj1 inj2 x` to choices in the second process and `inj2 x` to choices in the third process. We give here the main case referring to `Process+` (`Assur` is the function exchange the brackets of a disjoint union). This proof is by induction in the three processes, which in Agda turns into a recursive proof:

$$\begin{aligned}
& A\sqcup+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process+} \ \infty \ \{lu\} \ c_0) \\
& \quad (Q : \text{Process+} \ \infty \ \{lu\} \ c_1) \\
& \quad (Z : \text{Process+} \ \infty \ \{lu\} \ c_2) \\
& \rightarrow ((P \ \sqcup++ \ Q) \ \sqcup++ \ Z) \sqsubseteq+ \ \text{fmap+} \ \text{Ass}\text{ur} \ (P \ \sqcup++ \ (Q \ \sqcup++ \ Z)) \\
& A\sqcup+ \ P \ Q \ Z \text{.}[] \text{.nothing empty} = \text{empty}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{A}\square+ P Q Z . (\mathbf{Lab} P x :: l) m (\mathbf{extc} l . m (\mathbf{inj}_1 x) x_1) = \\
& \quad \mathbf{let} \\
& \quad \quad x' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty \mathbf{Ass}\uplus r (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{PE} P x))) \\
& \quad \quad x' = x_1 \\
& \quad \quad x_1' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty (\mathbf{Ass}\uplus r \circ \mathbf{inj}_1) ((\mathbf{PE} P x))) \\
& \quad \quad x_1' = \mathbf{lemFmap}\infty \mathbf{inj}_1 \mathbf{Ass}\uplus r (\mathbf{PE} P x) l m x' \\
& \quad \quad x_2' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{PE} P x))) \\
& \quad \quad x_2' = \mathbf{lemFmap}\infty \mathbf{R} \mathbf{inj}_1 \mathbf{inj}_1 (\mathbf{PE} P x) l m x_1' \\
& \quad \mathbf{in} \mathbf{extc} l m (\mathbf{inj}_1 (\mathbf{inj}_1 x)) x_2' \\
& \mathbf{A}\square+ P Q Z . (\mathbf{Lab} Q x :: l) m (\mathbf{extc} l . m (\mathbf{inj}_2 (\mathbf{inj}_1 x)) x_1) = \\
& \quad \mathbf{let} \\
& \quad \quad x_1'' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty \mathbf{Ass}\uplus r \\
& \quad \quad \quad (\mathbf{fmap}\infty \mathbf{inj}_2 (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{PE} Q x)))) \\
& \quad \quad x_1'' = x_1 \\
& \quad \quad x_1' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty (\mathbf{Ass}\uplus r \circ \mathbf{inj}_2) (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{PE} Q x))) \\
& \quad \quad x_1' = \mathbf{lemFmap}\infty \mathbf{inj}_2 \mathbf{Ass}\uplus r (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{PE} Q x)) l m x_1'' \\
& \quad \quad x_2' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty (\mathbf{Ass}\uplus r \circ \mathbf{inj}_2 \circ \mathbf{inj}_1) (\mathbf{PE} Q x)) \\
& \quad \quad x_2' = \mathbf{lemFmap}\infty \mathbf{inj}_1 (\mathbf{Ass}\uplus r \circ \mathbf{inj}_2) (\mathbf{PE} Q x) l m x_1' \\
& \quad \quad x_3' : \mathbf{Tr}\infty l m (\mathbf{fmap}\infty \mathbf{inj}_1 (\mathbf{fmap}\infty \mathbf{inj}_2 (\mathbf{PE} Q x))) \\
& \quad \quad x_3' = \mathbf{lemFmap}\infty \mathbf{R} \mathbf{inj}_2 \mathbf{inj}_1 (\mathbf{PE} Q x) l m x_2' \\
& \quad \mathbf{in} \mathbf{extc} l m (\mathbf{inj}_1 (\mathbf{inj}_2 x)) x_3' \\
& \mathbf{A}\square+ P Q Z . (\mathbf{Lab} Z y :: l) m (\mathbf{extc} l . m (\mathbf{inj}_2 (\mathbf{inj}_2 y)) x_1) = \\
& \quad \mathbf{extc} l m (\mathbf{inj}_2 y) (\mathbf{lemFmap}\infty \mathbf{inj}_2 (\mathbf{Ass}\uplus r \circ \mathbf{inj}_2) (\mathbf{PE} Z y) \\
& \quad \quad l m (\mathbf{lemFmap}\infty \mathbf{inj}_2 \mathbf{Ass}\uplus r (\mathbf{fmap}\infty \mathbf{inj}_2 (\mathbf{PE} Z y)) l m x_1)) \\
& \mathbf{A}\square+ P Q Z l m (\mathbf{intc} l . m (\mathbf{inj}_1 x) x_1) = \\
& \quad \mathbf{intc} l m (\mathbf{inj}_1 (\mathbf{inj}_1 x)) (\mathbf{A}\square\infty++ (\mathbf{PI} P x) Q Z l m x_1) \\
& \mathbf{A}\square+ P Q Z l m (\mathbf{intc} l . m (\mathbf{inj}_2 (\mathbf{inj}_1 x)) x_1) = \\
& \quad \mathbf{intc} l m (\mathbf{inj}_1 (\mathbf{inj}_2 x)) (\mathbf{A}\square+\infty+ P (\mathbf{PI} Q x) Z l m x_1) \\
& \mathbf{A}\square+ P Q Z l m (\mathbf{intc} l . m (\mathbf{inj}_2 (\mathbf{inj}_2 y)) x_1) = \\
& \quad \mathbf{intc} l m (\mathbf{inj}_2 y) (\mathbf{A}\square++\infty P Q (\mathbf{PI} Z y) l m x_1) \\
& \mathbf{A}\square+ P Q Z . [] . (\mathbf{just} (\mathbf{inj}_1 (\mathbf{inj}_1 (\mathbf{PT} P x)))) (\mathbf{terc} (\mathbf{inj}_1 x)) = \\
& \quad \mathbf{terc} (\mathbf{inj}_1 (\mathbf{inj}_1 x)) \\
& \mathbf{A}\square+ P Q Z . [] . (\mathbf{just} (\mathbf{inj}_1 (\mathbf{inj}_2 (\mathbf{PT} Q x))))
\end{aligned}$$

$$\begin{aligned}
& (\text{terc } (\text{inj}_2 (\text{inj}_1 x))) = \text{terc } (\text{inj}_1 (\text{inj}_2 x)) \\
\text{A}\square+ P Q Z . [] . (\text{just } (\text{inj}_2 (\text{PT } Z y))) (\text{terc } (\text{inj}_2 (\text{inj}_2 y))) = \\
& \text{terc } (\text{inj}_2 y)
\end{aligned}$$

Theorem 7.8.1 (Agda Theorem)

$$\begin{aligned}
\equiv \text{A}\square+ : & \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} \\
& (P : \text{Process}+ \infty \{lu\} \ c_0) \\
& (Q : \text{Process}+ \infty \{lu\} \ c_1) \\
& (Z : \text{Process}+ \infty \{lu\} \ c_2) \\
\rightarrow & ((P \square++ Q) \square++ Z) \equiv+ \text{fmap}+ \text{Ass}\uplus (P \square++ (Q \square++ Z))
\end{aligned}$$

Proof:

$$\equiv \text{A}\square+ P Q Z = (\text{A}\square+ P Q Z) , \text{A}\square+r P Q Z$$

ou

7.9 Proof of Law for Renaming Operator

We show that if we rename the labels in the process `terminate`, we obtain an equivalent process:

$$\begin{aligned}
\text{unitRenameLaw} : & \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\
\rightarrow & (A : \text{Label } lu \rightarrow \text{Label } lu) \\
\rightarrow & (P : \text{Process } i \{lu\} \ c_0) \\
\rightarrow & \text{Rename } A (\text{terminate } a) \sqsubseteq (\text{terminate } a) \\
\text{unitRenameLaw } a A P l m x = & x
\end{aligned}$$

$$\begin{aligned}
\text{unitRenameLawr} : & \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\
\rightarrow & (A : \text{Label } lu \rightarrow \text{Label } lu) \\
\rightarrow & (P : \text{Process } i \{lu\} \ c_0) \\
\rightarrow & (\text{terminate } a) \sqsubseteq \text{Rename } A (\text{terminate } a) \\
\text{unitRenameLawr } a A P l m x = & x
\end{aligned}$$

Theorem 7.9.1 (Agda Theorem)

$$\begin{aligned}
\equiv \text{unitRename} : & \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\
\rightarrow & (A : \text{Label } lu \rightarrow \text{Label } lu)
\end{aligned}$$

$$\begin{aligned} &\rightarrow (P : \text{Process } i \{lu\} \ c_0) \\ &\rightarrow \text{Rename } A \ (\text{terminate } a) \equiv (\text{terminate } a) \end{aligned}$$

Proof:

$$\equiv \text{unitRename } a \ A \ P = (\text{unitRenameLaw } a \ A \ P) , (\text{unitRenameLawr } a \ A \ P)$$

7.10 Proof of Laws for the Hiding Operator

The hiding operator replace chosen external transitions by internal one in order to hide them from other process. We show that hiding doesn't affect the process `terminate`:

$$\begin{aligned} \text{unitHideLaw} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\rightarrow (A : \text{Label } lu \rightarrow \text{Bool}) \\ &\rightarrow (P : \text{Process } i \{lu\} \ c_0) \\ &\rightarrow \text{Hide } A \ (\text{terminate } a) \sqsubseteq (\text{terminate } a) \\ \text{unitHideLaw } \{i\} \{c_0\} \ a \ A \ P \ l \ m \ q &= q \end{aligned}$$

$$\begin{aligned} \text{unitHideLawr} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\rightarrow (A : \text{Label } lu \rightarrow \text{Bool}) \\ &\rightarrow (P : \text{Process } i \{lu\} \ c_0) \\ &\rightarrow (\text{terminate } a) \sqsubseteq \text{Hide } A \ (\text{terminate } a) \\ \text{unitHideLawr } \{i\} \{c_0\} \ a \ A \ P \ l \ m \ q &= q \end{aligned}$$

Theorem 7.10.1 (Agda Theorem)

$$\begin{aligned} \equiv \text{unitHide} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0) \\ &\rightarrow (A : \text{Label } lu \rightarrow \text{Bool}) \\ &\rightarrow (P : \text{Process } i \{lu\} \ c_0) \\ &\rightarrow \text{Hide } A \ (\text{terminate } a) \equiv (\text{terminate } a) \end{aligned}$$

Proof:

$$\equiv \text{unitHide } a \ A \ P = (\text{unitHideLaw } a \ A \ P) , (\text{unitHideLawr } a \ A \ P)$$

Similarly we show that hiding of STOP gives the same process:

$$\text{stopHideLaw} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$$

```

      → (A : Label lu → Bool)
      → (P : Process i {lu} c₀)
      → Hide A (STOP c₀) ⊆ ((STOP c₀))
stopHideLaw {i} {c₀} a A (terminate x) .[] .nothing (tnode empty)
                                                    = tnode empty
stopHideLaw {i} {c₀} a A (terminate x₁) .(efq - :: l) m
                                                    (tnode (extc l .m () x₂))
stopHideLaw {i} {c₀} a A (terminate x) l m
                                                    (tnode (intc .l .m () x₂))
stopHideLaw {i} {c₀} a A (terminate x₁) .[] .(just (efq x))
                                                    (tnode (terc x)) = tnode (terc x)
stopHideLaw {i} {c₀} a A (node x) .[] .nothing (tnode empty)
                                                    = tnode empty
stopHideLaw {i} {c₀} a A (node x₁) .(efq - :: l) m
                                                    (tnode (extc l .m () x₂))
stopHideLaw {i} {c₀} a A (node x) l m (tnode (intc .l .m () x₂))
stopHideLaw {i} {c₀} a A (node x₁) .[] .(just (efq x))
                                                    (tnode (terc x)) = tnode (terc x)

```

Theorem 7.10.2 (Agda Theorem)

```

≡stopHide : {i : Size} {lu : LUniv} {c₀ : Choice} (a : ChoiceSet c₀)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c₀)
  → Hide A (STOP c₀) ≡ (STOP c₀)

```

Proof:

```

≡stopHide a A P = (stopHideLaw a A P) , (stopHideLaw a A P)

```

7.11 Proof of Termination and Unit for Interleaving Operator

We will show that `terminate` is the unit for the interleaving operator. A special case is that `terminate` interleaved with itself is `terminate` (modulo renaming of the result):

```

TerIntLaw : {lu : LUniv} {c₀ c₁ : Choice} (a : ChoiceSet c₀) (b : ChoiceSet c₁)
  → _⊔_ {lu} (terminate a ||| terminate b) (terminate ((a ,, b)))

```

TerIntLaw $\{lu\}\{c_0\}\{c_1\} a P l m q = q$

TerIntLawr : $\{lu : LUniv\}\{c_0 c_1 : Choice\} (a : ChoiceSet c_0) (b : ChoiceSet c_1)$
 $\rightarrow _ \sqsubseteq _ \{lu\} (\text{terminate } ((a ,, b))) (\text{terminate } a \parallel \text{terminate } b)$

TerIntLawr $\{lu\}\{c_0\}\{c_1\} a P l m q = q$

Theorem 7.11.1 (Agda Theorem)

$\equiv \text{TerInt+}$: $\{lu : LUniv\}\{c_0 c_1 : Choice\} (a : ChoiceSet c_0) (b : ChoiceSet c_1)$
 $\rightarrow _ \equiv _ \{lu\} (\text{terminate } a \parallel \text{terminate } b) (\text{terminate } ((a ,, b)))$

Proof:

$\equiv \text{TerInt+ } a b = \text{TerIntLaw } a b , \text{TerIntLawr } a b$

The other law states that *SKIP* is unit of interleaving operator (i.e. $SKIP \parallel P = P$). We can prove this property as follows:

unilntLaw : $\{lu : LUniv\}\{c_0 : Choice\} (a : ChoiceSet c_0)$
 $\rightarrow (P : Process \infty \{lu\} c_0)$
 $\rightarrow (\text{terminate } a \parallel P) \sqsubseteq \text{fmap } ((_, -) a) P$
unilntLaw $\{c_0\} a P l m q = q$

unilntLawr : $\{lu : LUniv\}\{c_0 : Choice\} (a : ChoiceSet c_0)$
 $\rightarrow (P : Process \infty \{lu\} c_0)$
 $\rightarrow \text{fmap } ((_, -) a) P \sqsubseteq (\text{terminate } a \parallel P)$
unilntLawr $\{c_0\} a P l m q = q$

Theorem 7.11.2 (Agda Theorem)

$\equiv \text{unilnt}$: $\{lu : LUniv\}\{c_0 : Choice\} (a : ChoiceSet c_0)$
 $\rightarrow (P : Process \infty \{lu\} c_0)$
 $\rightarrow (\text{terminate } a \parallel P) \equiv \text{fmap } ((_, -) a) P$

Proof:

$\equiv \text{unilnt } a P = (\text{unilntLaw } a P) , (\text{unilntLawr } a P)$

7.12 Proof of the Law of the Parallel Operator combined with terminate

We show that if the parallel composition of the process `terminate` with itself is modulo renaming the process `terminate`:

```

ter[-||-] : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
           (b : ChoiceSet c1) (A B : Label lu → Bool)
           → (terminate a [ A ]||[ B ] terminate b) ⊆
           fmap (λ x → a „ b) (terminate ((a „ b)))
ter[-||-] {c0} a P A B l m q = q

```

```

ter[-||-]r : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
           (b : ChoiceSet c1) (A B : Label lu → Bool)
           → fmap (λ x → a „ b) (terminate ((a „ b))) ⊆
           (terminate a [ A ]||[ B ] terminate b)
ter[-||-]r a P A B l m q = q

```

Theorem 7.12.1 (Agda Theorem)

```

≡ter[-||-] : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
           (b : ChoiceSet c1) (A B : Label lu → Bool)
           → (terminate a [ A ]||[ B ] terminate b)
           ≡ fmap (λ x → a „ b) (terminate ((a „ b)))

```

Proof:

```

≡ter[-||-] a b A B = (ter[-||-] a b A B) , (ter[-||-]r a b A B)

```

We have in this chapter defined the trace semantics of CSP in CSP-Agda, and adjusted it to the monadic setting. Since in CSP-Agda processes are monadic, we need to record, in case a process has terminated after following a trace, the return value of this process. We implemented this semantic, together with the corresponding refinement and equality relation, formally in CSP-Agda. We demonstrate the proof capabilities of CSP-Agda, by proving in CSP-Agda selected algebraic laws of CSP based on the trace semantics. The examples covered in this chapter are the laws of refinement, commutativity of interleaving, parallel, and external choice, the monad laws for the monadic extension of CSP, associativity of interleaving and external choice,

and laws for renaming, hiding, interleaving and parallel. All proofs and definitions have been typed checked in Agda.

Proofs of algebraic laws using trace semantics were carried out relatively smoothly. When introducing stable failures and FDI-semantics in the following this will not continue any more – direct proofs for these semantics will be quite difficult. However, we will overcome this problem by introducing in Chapter 10 strong bisimilarity and DRW bisimilarity. We will show there that these relations imply stable failure and FDI equivalence. Proofs using bisimilarity and DRW-bisimilarity will be relatively easy, and we will be able to show algebraic properties for stable failures and FDI semantics by first showing them using DRW bisimilarity or strong bisimilarity and then transferring them to stable failures and FDI semantics.

1412. Proof of the Law of the Parallel Operator combined with terminate

Chapter 8

Stable Failures Semantics

Trace semantics refers only to the visible (or observable) traces. It doesn't distinguish between external and internal choice. In particular, it does not tell what a process can refuse to do.

Take as an example the processes

$$\begin{aligned} &(a \longrightarrow P_1) \sqcap (b \longrightarrow P_2) \\ &\quad \text{and} \\ &(a \longrightarrow P_1) \sqcap (b \longrightarrow P_2) \end{aligned}$$

The first one allows an external choice a and then continues with P_1 , or an external choice b and continues with P_2 . The second one makes an internal choice to $a \longrightarrow P_1$ or $b \longrightarrow P_2$. In the first case it allows only external choice a followed by P_1 , and in the second case it allows only external choice b followed by P_2 . The traces of both processes are the same. But the second one can internally switch to $a \longrightarrow P_1$ or $b \longrightarrow P_2$, and in the first case refuse b , and in the second case refuse a . Stable failures semantics will distinguish between the two processes: The second process has two stable states (which means here it is a state without τ -transitions) $a \longrightarrow P_1$ and $b \longrightarrow P_2$, which can be reached by τ -transitions, and which refuse b and a , respectively. The process $(a \longrightarrow P_1) \sqcap (b \longrightarrow P_2)$ doesn't have states with the same properties¹.

The stable failures model has been developed to take care of this problem and to distinguish between external and internal choice. The stable failures model refers to a refusal set. A refusal set is a set X of external choices a process does not accept before following any other external choice. Stable failures in CSP are defined as pairs (t, X) , where $t \in \text{trace}(P)$ and X is a refusal of a process P' that one can obtain by following trace t , such that P' is stable. Roscoe and Schneider differ in their notion of stability, for

¹See Sect. 8.5 of Roscoe [1998] for the precise history of the stable failures model.

Schneider it means it has no τ -transitions, for Roscoe it means it has neither τ - nor \checkmark -transitions. We will later work mainly with Roscoe stability and use Schneider stability only as an auxiliary notion in proofs. Note that t records only the external choices of a sequence of transitions, there can be arbitrarily many τ -transitions involved. Note as well that there can be more than one process P' one can reach with the same trace t .

Refinement between two process in the stable failures semantics holds whenever a reversed subset relation holds between their sets of traces and stable failures (written as $failures(P)$):

$$P \sqsubseteq Q \text{ iff } traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P)$$

In this section, we represent the stable failures model in CSP-Agda. We note here already that direct proofs in CSP-Agda of algebraic properties with respect to stable failures equivalence are rather difficult. We will however prove those laws indirectly in Chapter 10 by proving the laws with respect to DRW-bisimilarity, and showing that this implies stable failures equivalence.

8.1 Traces with Next process

We first introduce a variant of the definition of a trace, in which we record as well the process we obtain after following that trace. More precisely, we define a predicate $(\text{TrP } l \ m \ P)$, which holds, if a process P after following a trace giving by a lists of labels l , reaches a possible next process m . So that P has trace l means that there is some m such that $(\text{TrP } l \ m \ P)$ holds. Since we have terminated processes, it might be that after following this trace we have terminated, therefore m can as well be a return value for the process. Combining the two possibilities, m is an element of $\text{Process} \ \infty \ c \ \uplus \ \text{ChoiceSet } c$. We define as well traces $(\text{TrP+ } l \ m \ P)$ and $(\text{TrP}\infty \ l \ m \ P)$ for processes in Process+ and $\text{Process}\infty$, respectively, similarly as we defined them in the traces model in Sect. 7. For elements of Process+ , the traces are the empty trace `empty`, external choice `extc`, internal choice `intc`, and traces resulting from a termination event `terc`. In the case of `terc`, the process has terminated, so m is `inj2 (PT P x)`. The definition of the extended traces in CSP-Agda is as follows:

```
data TrP+ {lu : LUniv}{c : Choice } : (l : List (Label lu))
  → Process ∞ {lu} c ⊔ ChoiceSet c
  → (P : Process+ ∞ {lu} c) → Set where
empty : {P : Process+ ∞ {lu} c} → TrP+ {lu} [] (inj1 (node P)) P
extc   : {P : Process+ ∞ {lu} c}
```

```

    → (l : List (Label lu))
    → (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (x : ChoiceSet (E P))
    → TrP∞ {lu} l tick (PE P x)
    → TrP+ {lu} (Lab P x :: l) tick P
intc  : {P : Process+ ∞ {lu} c}
    → (l : List (Label lu))
    → (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (x : ChoiceSet (I P))
    → TrP∞ {lu} l tick (PI P x)
    → TrP+ {lu} l tick P
terc  : {P : Process+ ∞ {lu} c}
    → (x : ChoiceSet (T P))
    → TrP+ {lu} [] (inj2 (PT P x)) P

```

For elements of $(\text{Process} \infty c)$, traces are the terminated trace **ter** for the terminated process, the empty trace **empty**, and traces (**tnode** *tr*) originating from a trace of a $(\text{Process+} \infty c)$:

```

data TrP {lu : LUniv}{c : Choice} : (l : List (Label lu))
  → Process ∞ {lu} c ⊔ ChoiceSet c
  → (P : Process ∞ {lu} c) → Set where
ter : (x : ChoiceSet c) → TrP {lu} [] (inj2 x) (terminate x)
empty : (x : ChoiceSet c) → TrP {lu} [] (inj1 (terminate x)) (terminate x)
tnode : {l : List (Label lu)}
  → {x : Process ∞ {lu} c ⊔ ChoiceSet c}
  → {P : Process+ ∞ {lu} c}
  → TrP+ {lu} {c} l x P
  → TrP {lu} l x (node P)

```

Finally, the traces for $\text{Process} \infty$ are the traces of the underlying Process :

```

TrP∞ : {lu : LUniv}{c : Choice} (l : List (Label lu))
  (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  (P : Process∞ ∞ {lu} c) → Set
TrP∞ {lu} {c} l tick P = TrP {lu} l tick (forcep P)

```



8.2 Stable Process

There are two different definitions, when a process is stable. According to Schneider [1999], it means that it cannot make any internal transitions, in CSP written as $P \downarrow = \neg (P \xrightarrow{\tau})$. According to Roscoe [1998], in addition, no \checkmark -transitions are allowed. We define a parametrised version, which depends on whether we choose the Roscoe version or not. We first define stability according to Schneider:

$\text{stableSch}_{\infty} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}_{\infty} \times \{lu\} \ c) \rightarrow \text{Set}$
 $\text{stableSch}_{\infty} \ P = \text{stableSch} \ (\text{forcep} \ P)$

$\text{stableSch}_{+} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}_{+} \times \{lu\} \ c) \rightarrow \text{Set}$
 $\text{stableSch}_{+} \ P = \neg (\text{ChoiceSet} \ (\text{I} \ P))$

$\text{stableSch} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process} \times \{lu\} \ c) \rightarrow \text{Set}$
 $\text{stableSch} \ (\text{terminate} \ x) = \top$
 $\text{stableSch} \ (\text{node} \ P) = \text{stableSch}_{+} \ P$

Note that not having an internal choice is defined as there not being any element in the internal choiceset. This is the correct notion since for the semantics all what is relevant is which external, internal choices and termination events a process can allow or perform; how they are indexed is a mere implementation detail. When defining operators of CSP, we used pattern matching on these choice sets. But equivalent choice sets should result in equivalent choice sets of the resulting processes.

Now we define a condition expressing that in case of Roscoe, no tick is allowed:

$\text{noTickIfRoscoe}_{+} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(\text{isRoscoe} : \text{Bool})$
 $(P : \text{Process}_{+} \times \{lu\} \ c) \rightarrow \text{Set}$
 $\text{noTickIfRoscoe}_{+} \ \text{false} \ P = \top$
 $\text{noTickIfRoscoe}_{+} \ \text{true} \ P = \neg (\text{ChoiceSet} \ (\top \ P))$

$\text{noTickIfRoscoe} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(\text{isRoscoe} : \text{Bool})$
 $(P : \text{Process} \times \{lu\} \ c)$
 $\rightarrow \text{Set}$
 $\text{noTickIfRoscoe} \ \text{false} \ (\text{terminate} \ x) = \top$
 $\text{noTickIfRoscoe} \ \text{true} \ (\text{terminate} \ x) = \perp$
 $\text{noTickIfRoscoe} \ \text{isRoscoe} \ (\text{node} \ Q) = \text{noTickIfRoscoe}_{+} \ \text{isRoscoe} \ Q$



Finally we give the parametrised definition of stable processes

$$\begin{aligned} \text{stableParametrized}\infty & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(\text{isRoscoe} : \text{Bool}) \\ & \quad (P : \text{Process}\infty \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stableParametrized}\infty \ b \ P & = \text{stableParametrized} \ b \ (\text{forcep} \ P) \end{aligned}$$

$$\begin{aligned} \text{stableParametrized}+ & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(\text{isRoscoe} : \text{Bool}) \\ & \quad (P : \text{Process}+ \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stableParametrized}+ \ \text{isRoscoe} \ P & = \text{stableSch}+ \ P \times \\ & \quad \text{noTickIfRoscoe}+ \ \text{isRoscoe} \ P \end{aligned}$$

$$\begin{aligned} \text{stableParametrized} & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(\text{isRoscoe} : \text{Bool}) \\ & \quad (P : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stableParametrized} \ \{c\} \ \text{isRoscoe} \ (\text{terminate} \ x) & = \neg (\text{T}' \ \text{isRoscoe}) \\ \text{stableParametrized} \ \{c\} \ b \ (\text{node} \ P) & = \text{stableParametrized}+ \ b \ P \end{aligned}$$

In this thesis most of the time we work with the Roscoe version, Schneider stability will be used as an auxiliary notion in proofs. Roscoe stability is defined as follows:

$$\begin{aligned} \text{stable}\infty & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}\infty \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stable}\infty \ P & = \text{stableParametrized}\infty \ \text{true} \ P \end{aligned}$$

$$\begin{aligned} \text{stable}+ & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stable}+ \ P & = \text{stableParametrized}+ \ \text{true} \ P \end{aligned}$$

$$\begin{aligned} \text{stable} & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ \text{stable} \ P & = \text{stableParametrized} \ \text{true} \ P \end{aligned}$$

Since in many cases we need just that there are no internal choices in case of stability, we define a function extracting this information:

$$\begin{aligned} \text{stabToNoInternal}+ & : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c) \\ & \quad (\text{stab} : \text{stable}+ \ P) \\ & \quad \rightarrow \neg (\text{ChoiceSet} \ (\text{I} \ P)) \\ \text{stabToNoInternal}+ \ P \ (\text{noInterCh} \ _, \ \text{notermEv}) \ \text{intChoice} & = \text{noInterCh} \ \text{intChoice} \end{aligned}$$

Furthermore, we want to combine the stability and notick properties to obtain parametrised stability:



```

stabSchNoTickIfRos2StablePar : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
    (isRoscoe : Bool)
    (stabSch : stableSch P)
    (notick : noTickIfRoscoe isRoscoe P)
    → stableParametrized isRoscoe P
stabSchNoTickIfRos2StablePar (terminate x) false stabSch notick ()
stabSchNoTickIfRos2StablePar (terminate x) true stabSch ()
stabSchNoTickIfRos2StablePar (node x) false stabSch notick = stabSch „ –
stabSchNoTickIfRos2StablePar (node x) true stabSch notick = stabSch „ notick

```

8.3 Refusal Sets

According to Schneider [1999] (Sect. 6.1), a set X is a refusal set for a process P , if after following an empty trace, i.e. after finitely many τ -transitions, we obtain a stable process, which does not allow any external choice transition in X . In CSP it is defined as follows ($P \downarrow$ means P is Schneider stable, i.e. P doesn't have any τ -transitions):

$$P \text{ ref } X = \exists P' \bullet (P \xRightarrow{\emptyset} P') \wedge (P' \downarrow) \wedge (\forall a \in X \bullet \neg (P' \xrightarrow{a}))$$

In the book by Roscoe [1998], this definition is modified: a process can refuse external choices only if it has no termination event, whereas in the book by Schneider [1999], a process which has termination events still can reject external choices.

For stable states, we have in case of Roscoe no termination events, so for stable failures the definition by Schneider does not need to be modified. However, in the failures/divergences/infinite traces model, non-stable failures are considered, therefore we parameterise the definition of refusal sets as for stability over whether we use the version of Roscoe or Schneider. If the parameter *isRoscoe* is **true**, for a refusal set no termination events are allowed, as in Roscoe, otherwise they are allowed.

We first define the notion of a process having no external choices in a given set of labels X :

```

NoExtChInX : {lu : LUniv}{c : Choice}(Q : Process+ ∞ c)
    (X : (Label lu) → Bool)
    → Set
NoExtChInX Q X = (e : ChoiceSet (E Q)) → ¬ (T'(X (Lab Q e)))

```



and that process doesn't allow termination events, if *isRoscoe* is **true**:

```

NoTicksIfIsRoscoe : {lu : LUniv}{c : Choice}(Q : Process+ ∞ {lu} c)
  (isRoscoe : Bool)
  → Set
NoTicksIfIsRoscoe Q isRoscoe = (tickIsIncl : T' isRoscoe)
  → ¬ (ChoiceSet (T Q))

```

Now we define what it means for a process to directly refuse all external events with labels in *X*:

```

data DRefusal+ {lu : LUniv}{c : Choice}(Q : Process+ ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  drefusal : (noextChInX : NoExtChInX Q X)
    (noTerm : NoTicksIfIsRoscoe Q isRoscoe)
    → DRefusal+ {lu}{c} Q isRoscoe X

DRefusal : {lu : LUniv}{c : Choice}(Q : Process ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) → Set
DRefusal {lu}{c} (terminate x) isRoscoe X = ¬ (T' isRoscoe)
DRefusal {lu}{c} (node x) isRoscoe X = DRefusal+ {lu}{c} x isRoscoe X

```

```

DRefusal∞ : {lu : LUniv}{c : Choice}(Q : Process∞ ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) → Set
DRefusal∞ {lu}{c} Q isRoscoe X = DRefusal {lu}{c} (forcep Q) isRoscoe X

```

Now we obtain the definition of refusal set of a process: it is the union of direct refusals of stable processes one can reach from a process by τ -transitions only:

```

data refusal {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)(isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP {lu}{c} [] (inj1 Q) P)
    (stab : stable Q)
    (drefuse : DRefusal Q isRoscoe X)
    → refusal P isRoscoe X

```

```

data refusal+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP+ {lu}{c} [] (inj1 Q) P)
    (stab : stable Q)
    (drefuse : DRefusal Q isRoscoe X)
    → refusal+ P isRoscoe X

data refusal∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)(isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP∞ {lu}{c} [] (inj1 Q) P)
    (stab : stable Q)
    (drefuse : DRefusal Q isRoscoe X)
    → refusal∞ P isRoscoe X

```

8.4 Stable Failures

The stable failures of a process P are list of labels l together with sets of labels X , such that after following a trace with labels l the process can reach a stable process, which refuses all events in X . This is written in CSP as

$$\exists P'' \bullet P \xRightarrow{tr} P'' \wedge P'' \downarrow \wedge P'' \text{ ref } X$$

Here $P'' \text{ ref } X$ expresses that P'' refuses all external events in X , even after making τ -transitions. Since a stable process cannot make any τ -transitions, for a stable process P we can define

$$P \text{ ref } X \Leftrightarrow \forall a \in X \bullet \neg (P' \xrightarrow{a})$$

So P refuses X after trace tr can be rewritten as

$$\exists P' \bullet P \xRightarrow{tr} P' \wedge P' \downarrow \wedge \forall a \in X \bullet \neg (P' \xrightarrow{a})$$

The definition in CSP-Agda is as follows:

```

data stableFailure+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)

```

```

(l : List (Label lu))
(isRoscoe : Bool)
(X : (Label lu) → Bool) : Set where
stableFp : (Q : Process ∞ {lu} c)
            (tr : TrP+ {lu}{c} l (injl Q) P)
            (stab : stable Q)
            (drefuse : DRefusal Q isRoscoe X)
            → stableFailure+ P l isRoscoe X

```

8.5 Refinement Relations

We define now two refinement relations: \sqsubseteq_{sf_1} expresses that the stable failures of the second process are stable failures of the first one (and we use the version of Roscoe here), and \sqsubseteq_{sf} is the conjunction of refinement for traces and of \sqsubseteq_{sf_1} . We first define \sqsubseteq_{sf_1} :

```

 $\sqsubseteq_{sf_1} : \{lu : LUniv\}\{c : Choice\} (P : Process \infty \{lu\} c)$ 
            $(Q : Process \infty \{lu\} c) \rightarrow Set$ 
 $\sqsubseteq_{sf_1} \{lu\}\{c\} P Q = (l : List (Label lu)) (X : (Label lu) \rightarrow Bool)$ 
            $\rightarrow stableFailure Q l \text{ true } X$ 
            $\rightarrow stableFailure P l \text{ true } X$ 

```

We define similar definitions for \sqsubseteq_{sf_1+} and between $Process+$ and $\sqsubseteq_{sf_1\infty}$ between $Process\infty$. The definition of \sqsubseteq_{sf} (with similar definitions of \sqsubseteq_{sf+} and $\sqsubseteq_{sf\infty}$ is as follows:

```

 $\sqsubseteq_{sf} : \{lu : LUniv\}\{c : Choice\} (P : Process \infty \{lu\} c)$ 
            $(Q : Process \infty \{lu\} c) \rightarrow Set$ 
 $P \sqsubseteq_{sf} Q = (P \sqsubseteq Q) \times (P \sqsubseteq_{sf_1} Q)$ 

 $=_{sf} : \{lu : LUniv\}\{c : Choice\} (P : Process \infty \{lu\} c)$ 
            $(Q : Process \infty \{lu\} c) \rightarrow Set$ 
 $P =_{sf} Q = (P \sqsubseteq_{sf} Q) \times (Q \sqsubseteq_{sf} P)$ 

```

8.6 Proofs for Stable Failures Semantics

It turns out that direct proofs using Stable Failures Semantics are very complicated except for simple properties. The reason is that in order to prove properties one needs to introduce lemma referring to any subprocesses obtained when evolving the processes. This is very cumbersome. It is much easier to carry out those proofs indirectly using forms of bisimilarity: we first show that processes are bisimilar and that the form of bisimilarity chosen implies equivalence with respect to stable failures semantics. Such proofs can be found in Chapter 9.

Chapter 9

Failures Divergences Infinite Traces Semantics

9.1 Motivation

The stable failures model as discussed in Sect. 8 does not analyse processes which can possibly diverge. Here a process is divergent, if it allows an infinite sequence of τ -transitions. The stable failures model ignores any divergent behaviour. For instance, if we consider the process Q , which has a τ -transition to itself and a τ transition to $STOP$. Process Q has the same traces as the $STOP$ process, namely only the empty one. The only stable failure process reachable from Q is the $STOP$ process, following an empty place, which is the same as the stable failures reachable from the $STOP$ process. So Q and $STOP$ are stable failure equivalent, but Q can diverge whereas the $STOP$ process cannot.

Therefore the *Failures/Divergences/Infinite Traces* model (*FDI*) of CSP has been developed to remedy this problem, where additional behaviours are introduced alongside information about failures. In this approach, we can identify a process P with the failures, divergences, and infinite traces that may be observed. Since this approach takes account of divergent, infinite behaviour and as well stable failures, it is more discriminating than the stable failures semantics.

The first component, referred to in this model, is the failures set, contains both stable failures and unstable failures. Unstable failures arise from divergent processes, which, because of their divergence, refuse everything.

The second component is *divergence*, which consists of all traces which lead to a divergent process.

The third component, infinite traces, is the set of all infinite sequences of events from Σ , a process can perform.

We note here already that direct proofs in CSP-Agda of algebraic properties with respect to FDI equivalence are rather difficult. We will however prove those laws indirectly in Chapter 10 by proving the laws with respect

to DRW-bisimilarity, and showing that this implies FDI equivalence.

9.2 Failures

As discussed in detail in Schneider [1999], Sect. 8.1, in the failures/divergence/infinite traces semantics there are two kind of failures: stable failures, as defined before, and unstable failures, arising from a divergent process. The union of those two sets is defined in CSP-Agda as follows:

```
data failure {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFail : stableFailure P l isRoscoe X
    → failure P l isRoscoe X
  divergentFailure : TraceDivergent ∞ c l P
    → failure P l isRoscoe X
```

Corresponding notions `failure∞` for `Process∞` and `failure+` for `Process+` are defined similarly.

9.3 Divergent Process

If a process P performs internal transitions forever, it cannot reach a stable state. In this case, the process P is called *divergent*, and written in CSP as $P \uparrow$.

There are two kinds of divergent behaviours. One is where there is a trace leading to a divergent state, i.e. a state where the process can only perform an infinite sequence of τ -transitions. The other one is an infinite trace, in which a process can have infinitely many observable events, but may as well fork off into a divergent behaviour. The generalisation of both is that we have an infinite sequence of τ -transitions starting with the process in question, and it is this notation which we formalise in CSP-Agda. We define *divergent* processes coinductively as follows:

```
record DivergentProcess∞ (i : Size){lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c) : Set where
  coinductive
  field
```



$\text{forcediv} : \{j : \text{Size} < i\} \rightarrow \text{DivergentProcess } j \{lu\} c (\text{forcep } P)$

$\text{data DivergentProcess } (i : \text{Size}) \{lu : \text{LUniv}\} (c : \text{Choice})$
 $\quad : (P : \text{Process} \infty \{lu\} c) \rightarrow \text{Set where}$
 $\text{div} : (P : \text{Process} + \infty c) (\text{divP} : \text{DivergentProcess} + i c P)$
 $\quad \rightarrow \text{DivergentProcess } i c (\text{node } P)$

$\text{data DivergentProcess} + (i : \text{Size}) \{lu : \text{LUniv}\} (c : \text{Choice})$
 $\quad (P : \text{Process} + \infty \{lu\} c) : \text{Set where}$
 $\text{div} + : (\text{int} : \text{ChoiceSet } (I P))$
 $\quad (\text{divP} : \text{DivergentProcess} \infty i c (PI P \text{int}))$
 $\quad \rightarrow \text{DivergentProcess} + i c P$

9.4 Divergent Traces

The traces of process considered as divergent traces, if after finite sequence of event a divergent state reached. In CSP tr is a divergent trace for process P if there exists a Q s.t.

$$P \xRightarrow{tr} Q \wedge (Q \uparrow)$$

We define *divergent* traces as follows:

$\text{data TraceDivergent} + (i : \text{Size}) \{lu : \text{LUniv}\} (c : \text{Choice})$
 $\quad (l : \text{List } (\text{Label } lu)) (P : \text{Process} + \infty \{lu\} c) : \text{Set where}$
 $\text{trdiv} : (Q : \text{Process} \infty \{lu\} c) (\text{trp} : \text{TrP} + \{lu\} \{c\} l (\text{inj}_1 Q) P)$
 $\quad (\text{divp} : \text{DivergentProcess } i c Q)$
 $\quad \rightarrow \text{TraceDivergent} + i c l P$

$\text{data TraceDivergent } (i : \text{Size}) \{lu : \text{LUniv}\} (c : \text{Choice})$
 $\quad (l : \text{List } (\text{Label } lu))$
 $\quad (P : \text{Process} \infty \{lu\} c) : \text{Set where}$
 $\text{trdiv} : (Q : \text{Process} \infty \{lu\} c) (\text{trp} : \text{TrP } \{lu\} \{c\} l (\text{inj}_1 Q) P)$
 $\quad (\text{divp} : \text{DivergentProcess } i c Q)$
 $\quad \rightarrow \text{TraceDivergent } i c l P$



```

data TraceDivergent $\infty$  (i : Size){lu : LUniv}(c : Choice)
  (l : List (Label lu))
  (P : Process $\infty$   $\infty$  {lu} c) : Set where
  trdiv : (Q : Process  $\infty$  {lu} c) (trp+ : TrP $\infty$  {lu} {c} l (injl Q) P)
    (divp : DivergentProcess i c Q)
    → TraceDivergent $\infty$  i c l P

```

9.5 Infinite Traces

In order to introduce the notion of infinite traces, we first need the notion of infinite streams of labels. This definition is the coinductive form of a list, except that we don't have an empty list. We repeat the definition of streams from Subsect. 3.2.5 in Agda:

```

record Stream {i : Size} (X : Set) : Set where
  coinductive
  field
  head : X
  tail : {j : Size < i} → Stream {j} X

```

The infinite traces for a `Process+` are given in the following. This definition is inductive in case of internal choices, since we allow only finitely many internal choices before an external choice is chosen, whereas coinductive in the external choices. The difference is recorded in the different types `infTr ∞` and `infTr ∞ '`:

```

data infTr+ {i : Size} {lu : LUniv}{c : Choice}
  : (l : Stream { $\infty$ } (Label lu))
    → (P : Process+  $\infty$  {lu} c) → Set where
  extc : {P : Process+  $\infty$  {lu} c}
    → (l : Stream { $\infty$ } (Label lu))
    → (x : ChoiceSet (E P))
    → (T' (head l == Lab P x))
    → infTr $\infty$  {i} {lu}{c} (tail l) (PE P x)
    → infTr+ {i} {lu}{c} l P
  intc : {P : Process+  $\infty$  {lu} c}
    → (l : Stream { $\infty$ } (Label lu))
    → (x : ChoiceSet (I P))

```

$$\begin{aligned} &\rightarrow \text{infTr}\infty' \{i\} \ l \ (\text{PI } P \ x) \\ &\rightarrow \text{infTr}+ \{i\} \ l \ P \end{aligned}$$

For **Process**, we just refer to the notion **infTr+**, however we don't allow a termination event:

```
data infTr {i : Size} {lu : LUniv} {c : Choice} :
  (l : Stream {∞} (Label lu)) →
  (P : Process ∞ {lu} c) → Set where
tnode : {l : Stream {∞} (Label lu)}
  → {P : Process+ ∞ {lu} c}
  → infTr+ {i} {lu} {c} l P
  → infTr {i} {lu} l (node P)
```

Finally, we define the notion for **Process∞**. We have one notion **infTr∞**, which coinductively refers to a process of smaller size. The notion **infTr∞'** keeps the size and is inductive. The definition in Agda is as follows:

```
record infTr∞ {i : Size} {lu : LUniv} {c : Choice}
  (l : Stream {∞} (Label lu))
  (P : Process∞ ∞ {lu} c) : Set where
  coinductive
  field
    forcetP : {j : Size< i} → infTr {j} l (forceP P)

infTr∞' : {i : Size} {lu : LUniv} {c : Choice}
  (l : Stream {∞} (Label lu))
  (P : Process∞ ∞ {lu} c) → Set
infTr∞' {i} {lu} {c} l P = infTr {i} l (forceP P)
```

We note here that in Agda a mutual coinductive/inductive definition will be interpreted as always-eventually ($\nu \mu$) rather than eventually always ($\mu \nu$).

9.6 Refinement Relations

We define now four refinement relations: first we include the refinement notion of trace. This is needed, since constructively not every trace will be part of one of the other notions. Next we define $\sqsubseteq_{\text{fdi}_1}$, which expresses that the divergent traces of the second process are divergent traces of the first one. Then we have $\sqsubseteq_{\text{fdi}_2\text{ros}}$ expressing that the refusals of the second

process are refusals of the first one (and we use the version of Roscoe here). Then we define $\sqsubseteq_{\text{fdi}_3}$ expressing refinement with respect to infinite traces. Finally we define \sqsubseteq_{fdi} as being the conjunction of refinement for traces and of the previous three refinement relations.

$$\begin{aligned} \sqsubseteq_{\text{fdi}_1} &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\ &\quad (Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ \sqsubseteq_{\text{fdi}_1} \{lu\} \{c\} P Q &= (l : \text{List} (\text{Label } lu)) \\ &\quad \rightarrow \text{TraceDivergent} \infty c l Q \\ &\quad \rightarrow \text{TraceDivergent} \infty c l P \end{aligned}$$

$$\begin{aligned} \sqsubseteq_{\text{fdi}_2\text{ros}} &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\ &\quad (Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ \sqsubseteq_{\text{fdi}_2\text{ros}} \{lu\} \{c\} P Q &= (l : \text{List} (\text{Label } lu)) \\ &\quad (X : (\text{Label } lu) \rightarrow \text{Bool}) \\ &\quad \rightarrow \text{failure } Q l \text{ true } X \\ &\quad \rightarrow \text{failure } P l \text{ true } X \end{aligned}$$

$$\begin{aligned} \sqsubseteq_{\text{fdi}_3} &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\ &\quad (Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ \sqsubseteq_{\text{fdi}_3} \{lu\} \{c\} P Q &= (l : \text{Stream} \{\infty\} (\text{Label } lu)) \\ &\quad (tr : \text{infTr} \{\infty\} l Q) \\ &\quad \rightarrow \text{infTr} \{\infty\} l P \end{aligned}$$

$$\begin{aligned} \sqsubseteq_{\text{fdi}} &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\ &\quad (Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set} \\ P \sqsubseteq_{\text{fdi}} Q &= (((P \sqsubseteq Q) \times (P \sqsubseteq_{\text{fdi}_1} Q)) \times (P \sqsubseteq_{\text{fdi}_2\text{ros}} Q)) \times (P \sqsubseteq_{\text{fdi}_3} Q) \end{aligned}$$

$$\begin{aligned} \equiv_{\text{fdi}} &: \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P Q : \text{Process} \infty \{lu\} c_0) \rightarrow \text{Set} \\ P \equiv_{\text{fdi}} Q &= (P \sqsubseteq_{\text{fdi}} Q) \times (Q \sqsubseteq_{\text{fdi}} P) \end{aligned}$$

9.7 Proofs in Failures Divergences Infinite Traces Semantics

As for stable failure semantics, carrying proofs directly in the failures divergences and infinite traces mode is more complex than expected. For instance, if one wants to show commutativity of \sqsubseteq , one needs to first investigate the form of processes one obtains after following a trace starting with $(Q \sqsubseteq P)$, and needs to show that one obtains the same process, but commuted, if one starts with $(P \sqsubseteq Q)$. However, after an external choice these processes may have a different form, namely $(\text{fmap}\infty \text{ inj}_i P)$, which one needs to take care of. So one needs to introduce different lemmas for all different forms of processes one obtains. Then one needs to show that the two processes obtained after a trace have the same properties regarding being stable, divergent, refusal sets, and infinite traces. This is very tedious. Therefore, in the following we will present only the straightforward proof that refinement with respect to stable failures semantics is a partial order, which can be carried out more easily.

In Chapter 10 we will then show laws regarding stable failure semantics in an indirect way, which is much easier: we show that divergence-respecting weak bisimilarity and strong bisimilarity imply equivalence with respect to stable failures semantics and then show that certain algebraic laws hold with respect to one of these two bisimilarity relations.

We show now that the refinement relations \sqsubseteq , $\sqsubseteq_{\text{fdi}_1}$, $\sqsubseteq_{\text{fdi}_2\text{ros}}$, $\sqsubseteq_{\text{fdi}_3}$ and \sqsubseteq_{fdi} are reflexive, antisymmetric, and transitive, i.e. fulfil the following laws (where \sqsubseteq is one of these relations and \equiv the corresponding equality relation):

$$\begin{aligned} P &\sqsubseteq P \\ P_0 &\sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 \equiv P_1 \\ P_0 &\sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2 \end{aligned}$$

For the first three of the above relations, the definition is given by stating that if the second process fulfils a certain property (e.g. that tr is a trace) the first process fulfils it as well. They are equivalent if refinement goes in both directions. This implies immediately reflexivity, antisymmetry, and transitivity. Furthermore, \sqsubseteq_{fdi} is the conjunction of \sqsubseteq , $\sqsubseteq_{\text{fdi}_1}$ and $\sqsubseteq_{\text{fdi}_2}$, and therefore (omitting similar proofs of the above properties for \sqsubseteq and $\sqsubseteq_{\text{fdi}_2}$) we obtain reflexivity, antisymmetry, and transitivity for \sqsubseteq_{fdi} as well:

$$\text{refl}\sqsubseteq_{\text{fdi}_1} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$$

$$\begin{aligned} & (P : \text{Process} \infty \{lu\} \ c) \rightarrow P \sqsubseteq_{\text{fdi}_1} P \\ \text{refl}_{\sqsubseteq_{\text{fdi}_1}} \ P \ l \ \text{divp} &= \text{divp} \end{aligned}$$

$$\begin{aligned} \text{antiSym}_{\sqsubseteq_{\text{fdi}_1}} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \\ & \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow P \sqsubseteq_{\text{fdi}} Q \\ & \rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q \\ \text{antiSym}_{\sqsubseteq_{\text{fdi}_1}} \ P \ Q \ PQ \ QP &= PQ \ \text{,,} \ QP \end{aligned}$$

$$\begin{aligned} \text{trans}_{\sqsubseteq_{\text{fdi}_1}} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \\ & (Q : \text{Process} \infty \{lu\} \ c) \\ & (R : \text{Process} \infty \{lu\} \ c) \\ & \rightarrow P \sqsubseteq_{\text{fdi}_1} Q \rightarrow Q \sqsubseteq_{\text{fdi}_1} R \rightarrow P \sqsubseteq_{\text{fdi}_1} R \\ \text{trans}_{\sqsubseteq_{\text{fdi}_1}} \ P \ Q \ R \ PQ \ QR \ l \ \text{divp} &= PQ \ l \ (QR \ l \ \text{divp}) \end{aligned}$$

$$\begin{aligned} \text{refl}_{\sqsubseteq_{\text{fdi}}} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \rightarrow P \sqsubseteq_{\text{fdi}} P \\ \text{refl}_{\sqsubseteq_{\text{fdi}}} \ P &= ((\text{refl}_{\sqsubseteq} \ P \ \text{,,} \ \text{refl}_{\sqsubseteq_{\text{fdi}_1}} \ P) \ \text{,,} \ \text{refl}_{\sqsubseteq_{\text{fdi}_2}} \ P) \ \text{,,} \ \text{refl}_{\sqsubseteq_{\text{fdi}_3}} \ P \end{aligned}$$

$$\begin{aligned} \text{antiSym}_{\sqsubseteq_{\text{fdi}}} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \\ & \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow P \sqsubseteq_{\text{fdi}} Q \\ & \rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q \\ \text{antiSym}_{\sqsubseteq_{\text{fdi}}} \ P \ Q \ PQ \ QP &= PQ \ \text{,,} \ QP \end{aligned}$$

$$\begin{aligned} \text{trans}_{\sqsubseteq_{\text{fdi}}} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \\ & (Q : \text{Process} \infty \{lu\} \ c) \\ & (R : \text{Process} \infty \{lu\} \ c) \\ & \rightarrow P \sqsubseteq_{\text{fdi}} Q \rightarrow Q \sqsubseteq_{\text{fdi}} R \rightarrow P \sqsubseteq_{\text{fdi}} R \\ \text{trans}_{\sqsubseteq_{\text{fdi}}} \ P \ Q \ R \ (((PQ \ \text{,,} \ PQfdi_1) \ \text{,,} \ PQfdi_2) \ \text{,,} \ PQfdi_3) \\ & (((QR \ \text{,,} QRfdi_1) \ \text{,,} QRfdi_2) \ \text{,,} QRfdi_3) \\ &= (((\text{trans}_{\sqsubseteq} \ P \ Q \ R \ PQ \ QR \\ & \ \text{,,} \ \text{trans}_{\sqsubseteq_{\text{fdi}_1}} \ P \ Q \ R \ PQfdi_1 \ QRfdi_1) \\ & \ \text{,,} \ \text{trans}_{\sqsubseteq_{\text{fdi}_2}} \ P \ Q \ R \ PQfdi_2 \ QRfdi_2) \\ & \ \text{,,} \ \text{trans}_{\sqsubseteq_{\text{fdi}_3}} \ P \ Q \ R \ PQfdi_3 \ QRfdi_3) \end{aligned}$$

Chapter 10

Bisimulation

The notions of determining process equivalence in CSP are defined via traces, failures, divergences etc., and therefore not directly based on the underlying transition system. Other process algebras like CCS define the underlying meaning of processes directly by using the underlying labelled transition system (*LTS*). That approach allows to decide equivalence by deciding whether the corresponding LTSs are essentially the same. Several equivalences over LTSs for what it means to be “essentially the same” been suggested, which are called bisimilarities.

As discussed in section 9.7, proofs in FDI semantics and in stable failures semantics turn out to be more complex than expected. Consider a proof of commutativity of external choice \square . In a direct proof of it, one first needs to investigate the form of processes one obtains after following a trace starting with $(Q \square P)$, and then one needs to show that one obtains the same process, but commuted, if one starts with $(P \square Q)$. However, after an external choice these processes have a different form, for instance $(\text{fmap} \infty \text{ inj}_i P)$, which one needs to take care of. Then one needs to show that the two processes obtained after a trace have the same properties regarding being stable, divergent, and refusal sets.

In this chapter we present a more elegant way by defining *strong* and *divergence-respecting weak bisimilarity* (DRW). We will show algebraic properties hold with respect to one of these relations and that these relations imply equivalence with respect to stable failure and FDI semantics. This way we get a proof of algebraic laws with respect to stable failure and FDI semantics in an indirect way.

We will first introduce strong (Sect. 10.1) and divergence-respecting weak bisimilarity (Sect. 10.2) in CSP Agda, following the definitions in Roscoe [2010, 1998]. Then we prove reflexivity for strong (Sect. 10.3) and divergent respecting weak bisimilarity (Sect. 10.4), and symmetry of DRW bisimilarity (Sect. 10.5). In the next section 10.6, we carry out the proof, that

strong bisimilarity implies DRW bisimilarity in Agda. Furthermore, we prove strong and weak bisimilarity (and therefore as well divergence respecting weak bisimilarity) imply trace equivalence. Then, we prove in Sect. 10.8 the key lemma Lemma 2.4.6, and then obtain that bisimilarity implies stable failures equivalence (Sect. 10.10) and and FDI equivalence (Sect. 10.11). In Sect. 10.12, we prove selected algebraic laws indirectly using strong bisimilarity: Commutativity of the external choice operator (Sect. 10.12.1), and of the interleaving operator (Sect. 10.12.2), and of the first and third monadic law (Sect. 10.13). We will discuss the problems with respect to the second monadic law.

10.1 Defining Strong Bisimilarity for CSP-Agda

In Sect 2.4.5 we introduced the notion of strong bisimilarity. In short, two processes are strongly bisimilar, if they have the same set of events, and for each set of events the processes we obtained are again strongly bisimilar. This recursive definition needs to be considered coinductive, since processes might proceed for ever.

In CSP-Agda we define strong bisimilarity directly in a coinductive way in the same way as processes were defined in section 5.3, and obtain the largest bisimilarity relation. In the following we define the predicate `Bisims` expressing that two processes are bisimilar. We define as well `Bisims+` and `Bisims ∞` for processes in `Process+` and `Process ∞` respectively.

Since processes in CSP-Agda are described in a monadic way, we need to check for bisimilar processes that in case they have terminated or not terminate, the results obtained by them are equal.

For an element of `Process` we get the following definition of bisimilarity:

- Terminated processes are strongly bisimilar, iff their return values are the same, and we denote the proof by `eqterminate`.
- The notion of bisimilarity for non-terminated processes refers to the corresponding notion of bisimilarity for `Process+`, and the resulting proof will be denoted by `eqnode`.

The corresponding definition for the processes for `Process` is as follows:

```
data Bisims {i : Size}{lu : LUniv}{c : Choice} :
  (P P' : Process  $\infty$  {lu} c) → Set where
```

```

eqterminate : { a : ChoiceSet c }
              → Bisims {i} (terminate a) (terminate a)
eqnode : { Q Q' : Process+ ∞ {lu} c } → Bisims+ {i} Q Q'
        → Bisims {i} (node Q) (node Q')

```

In case of **Process+**, we obtain that two processes are strongly bisimilar, iff:

- For every external choice of the first process, we get an external choice for the second process, where the label for these processes are same, and the resulting process are bisimilar.
- In case the first one has an internal choice, the second one has an internal choice as well, and the resulting two processes are bisimilar.
- If the first process has a termination event, then the second one needs to have as well an termination event, and the returned values must be equal.
- The reverse direction (from 2nd to first process) holds as well.

We obtain the following definition of the bisimilarity for **Process+**:

```

record Bisims+ {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c) : Set where
  coinductive
  field
    bisim2E    : (e : ChoiceSet (E P)) → ChoiceSet (E P')
    bisimELab  : (e : ChoiceSet (E P))
                  → Lab P e ≡ Lab P' (bisim2E e)
    bisimENext : (e : ChoiceSet (E P))
                  → Bisims∞ {i} (PE P e) (PE P' (bisim2E e))
    bisim2I    : (int1 : ChoiceSet (I P)) → ChoiceSet (I P')
    bisimINext : (int1 : ChoiceSet (I P))
                  → Bisims∞ {i} (PI P int1) (PI P' (bisim2I int1))
    bisim2T    : (t : ChoiceSet (T P)) → ChoiceSet (T P')
    bisim2TEq  : (t : ChoiceSet (T P))
                  → PT P t ≡ PT P' (bisim2T t)
    bisim2Er   : (e : ChoiceSet (E P')) → ChoiceSet (E P)
    bisimELabr : (e : ChoiceSet (E P'))
                  → Lab P' e ≡ Lab P (bisim2Er e)
    bisimENextr : (e : ChoiceSet (E P'))

```

$$\begin{aligned}
 & \rightarrow \text{Bisims}_{\infty} \{i\} (\text{PE } P (\text{bisim2Er } e)) (\text{PE } P' e) \\
 \text{bisim2lr} & : (int1 : \text{ChoiceSet } (I \ P')) \rightarrow \text{ChoiceSet } (I \ P) \\
 \text{bisimlNextr} & : (int1 : \text{ChoiceSet } (I \ P')) \\
 & \rightarrow \text{Bisims}_{\infty} \{i\} (\text{PI } P (\text{bisim2lr } int1)) (\text{PI } P' int1) \\
 \text{bisim2Tr} & : (t : \text{ChoiceSet } (T \ P')) \rightarrow \text{ChoiceSet } (T \ P) \\
 \text{bisim2TEqr} & : (t : \text{ChoiceSet } (T \ P')) \\
 & \rightarrow \text{PT } P' t \equiv \text{PT } P (\text{bisim2Tr } t)
 \end{aligned}$$

Finally, bisimilarity for Process_{∞} is bisimilarity of the underlying Process :

```

record Bisims∞ {i : Size} {lu : LUniv} {c : Choice}
  (P P' : Process∞ ∞ {lu} c) : Set where
  coinductive
  field
  forceB : {j : Size < i} → Bisims {j} {lu} (forceP P) (forceP P')
    
```

10.2 Defining Divergence-Respecting Weak Bisimilarity for CSP-Agda

Strong bisimilarity as an equality on CSP processes is too strong. The processes $\tau \rightarrow P$ and P should be in CSP equivalent, but they are usually not strongly bisimilar. For instance, if P has no internal choice we have that $\tau \rightarrow P$ has a τ -transition, whereas P doesn't. In order to fix this one needs to move to weak bisimilarity, which essentially ignores finitely many τ -transitions.

However, there is still problem: Weak bisimilarity identifies processes which differ only in divergent processes. For instance the divergent process DIV , which has only a τ -transition to itself, and the process $STOP$, which has no transitions, are weakly bisimilar, since the relation $\{(DIV, STOP)\}$ is a weak bisimilarity. For DIV we have $DIV \xRightarrow{s} P \Leftrightarrow (P = DIV \wedge s = [])$ and for $STOP$ we have $STOP \xRightarrow{s} P \Leftrightarrow (P = STOP \wedge s = [])$. However DIV and $STOP$ are neither stable failures equivalent nor FDI-equivalent. One can show that weak bisimilarity implies trace equivalence. So weak bisimilarity only implies trace equivalence, but not stable failure or FDI equivalence.

In order to fix this one introduces *divergence-respecting weak bisimilarity*, which is a slight strengthening of weak bisimilarity, by demanding in addition that for two weakly bisimilar processes, if one is divergent, then the other

is divergent as well. The definitions of weak and divergence-respecting weak bisimilarity in CSP can be found in Sect.2.4.5.

In Agda we first use the definition of divergent process. We repeat the definition, which was already given in 9.3, for convenience: a process is divergent, if it has a infinitely many τ -transitions as given by a coinductive definition:

```
record DivergentProcess $\infty$  (i : Size){lu : LUniv}(c : Choice)
  (P : Process $\infty$   $\infty$  {lu} c) : Set where
  coinductive
  field
  forcediv : {j : Size< i}  $\rightarrow$  DivergentProcess j {lu} c (forcep P)

data DivergentProcess (i : Size){lu : LUniv}(c : Choice)
  : (P : Process  $\infty$  {lu} c)  $\rightarrow$  Set where
  div : (P : Process+  $\infty$  c) (divP : DivergentProcess+ i c P)
     $\rightarrow$  DivergentProcess i c (node P)

data DivergentProcess+ (i : Size){lu : LUniv}(c : Choice)
  (P : Process+  $\infty$  {lu} c) : Set where
  div+ : (int : ChoiceSet (I P))
    (divP : DivergentProcess $\infty$  i c (PI P int))
     $\rightarrow$  DivergentProcess+ i c P
```

Since we are using constructive logic, we need to deal with the fact that the negation of divergence doesn't imply that a process eventually becomes stable, a property which we need in order to prove that DRW-bisimilarity implies stable failure equivalence. In order to fix this we introduce a positive notion of non-divergence expressing that a process eventually becomes stable. We need as well for the property that for any subprocess obtained when following τ -transitions we can decide whether there is a further τ -transition or not, in order to be able to determine a stable subprocess. The definition is in Agda as follows:

```
record NonDivergent $\infty$  {i : Size}{lu : LUniv}{c : Choice}
  (P : Process $\infty$   $\infty$  {lu} c) : Set where
  inductive
  field
  forceND : {j : Size< i}  $\rightarrow$  NonDivergent {j} (forcep P)
```

```

NonDivergent : {i : Size}{lu : LUniv}{c : Choice}
               (P : Process ∞ {lu} c) → Set
NonDivergent (terminate x) = ⊤
NonDivergent {i} (node Q) = NonDivergent+ {i} Q

data NonDivergent+ {i : Size}{lu : LUniv}{c : Choice}
                 (P : Process+ ∞ {lu} c) : Set where
  nondiv : ((int1 : ChoiceSet (I P)) → NonDivergent∞ {i} (PI P int1))
    → (chemptyornot : ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P) ))
    → NonDivergent+ {i} P

```

Note this extra condition *chemptyornot*, namely that it is decidable whether there is an internal choice or not, and in case of yes we can determine an index for an internal choice. We need this in order to find for a non-divergent process a stable process which is reachable from it by τ -transition – if the process has a τ -transition we choose one (as given by *chemptyornot*), and if it hasn't we are at least stable in the sense of Schneider (Roscoe requires as well that there are no termination events). In general *chemptyornot* does not hold, it holds usually for explicitly given processes.

Since in CSP-Agda processes are monadic, we need to make sure that the return values of terminated processes are the same. However, since in weak bisimilarity we ignore finitely many τ -transitions, we need to treat a process having as only event a termination event as being equivalent to a terminated process with the same return value. Therefore we define more general what it means to be a termination equivalent process:

- A process is termination equivalent with return value a , if all it can do is having finitely many τ transitions after which it can always do either a termination event with return value a , or it becomes the terminated process with return value a .

In addition we need that a termination equivalent process either has a silent transition, or has no silent transition and a termination event – if it had neither, it would be the STOP process which is not termination equivalent; if it had both, it could do two different things at the same time.

- The process *terminate* a is weakly bisimilar to Q , iff Q is termination equivalent with return value a .

The resulting code for termination equivalence in Agda is as follows:

```

TerminateEquivalent $\infty$  : {lu : LUniv}{c : Choice}(a : ChoiceSet c)
    (P : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  Set
TerminateEquivalent $\infty$  a P = TerminateEquivalent a (forcep P)

data TerminateEquivalent {lu : LUniv}{c : Choice}(a : ChoiceSet c)
    : (P : Process  $\infty$  {lu} c)  $\rightarrow$  Set where
    termeqterm : TerminateEquivalent a (terminate a)
    termeqnode : {P : Process+  $\infty$  {lu} c}
        (terequivP : TerminateEquivalent+ a P)
         $\rightarrow$  TerminateEquivalent a (node P)

record TerminateEquivalent+ {lu : LUniv}{c : Choice}(a : ChoiceSet c)
    (P : Process+  $\infty$  {lu} c) : Set where
    inductive
    field
        noExtChoice      : (e : ChoiceSet (E P))  $\rightarrow$   $\perp$ 
        onlyIntChoice    : (i : ChoiceSet (I P))
             $\rightarrow$  TerminateEquivalent $\infty$  a (PI P i)
        termIsa          : (t : ChoiceSet (T P))  $\rightarrow$  a  $\equiv$  PT P t
        hasTauOrTickNoTau : ChoiceSet (I P)  $\uplus$ 
            ( $\neg$  (ChoiceSet (I P))  $\times$  ChoiceSet (T P))

```

We note here the condition `hasTauOrTickNoTau`. It expresses that we have an explicit internal choice, which we could use in order to find a Schneider stable process reachable by τ -transitions only, or there is no internal choice, in which case we have a termination event which by `termIsa` must have return value a . This condition allows to find by following τ -transition a process which has no τ -transition but a \checkmark -event.

We can now define that two processes are DRW-bisimilar, iff:

- If the first process is the terminated process with return value a , then the other process is termination equivalent with return value a . The corresponding proof is denoted by `eqterminate`.
 - The same, with the two processes interchanged; the constructor for this proof is called `eqterminater`.
 - Two non-terminated process are DRW-bisimilar, if the corresponding elements of `Process+` are so.
-

We obtain the following definition of bisimilarity for `Process`:

```
data Bisimw {i : Size}{lu : LUniv}{c : Choice}
  : (P P' : Process ∞ {lu} c) → Set where
eqterminate : {a : ChoiceSet c} → {P' : Process ∞ {lu} c}
  (terequiv : TerminateEquivalent a P')
  → Bisimw {i} (terminate a) P'
eqterminater : {a : ChoiceSet c} → {P : Process ∞ {lu} c}
  (terequiv : TerminateEquivalent a P)
  → Bisimw {i} P (terminate a)
eqnode : {Q Q' : Process+ ∞ {lu} c}
  (bisimQQ' : Bisimw+ {i} Q Q')
  → Bisimw {i} (node Q) (node Q')
```

In case of `Process+`, two processes P and P' are DRW-bisimilar, iff the following holds:

- If the process P is divergent, then P' is as well divergent, given by field `bisimdiv` below.
- In case P is non-divergent, then P' is non-divergent, given by field `nondiv+` below.
- If P has an external choice with label l resulting in process Q , then P' has a trace $[l]$ resulting in a process Q' which is DRW-bisimilar to Q . This property is given by the three fields `bisimEP'`, `bisimEtr`, and `bisimEnext`.
- If P has an internal choice resulting in process Q , then P' has a trace $[]$ resulting in a process Q' which is DRW-bisimilar to Q . This property is given by the three fields `bisimIP'`, `bisimItr`, and `bisimInext`.
- If P has a termination event with return value a , then P' has a trace $[]$ ending in a termination event with return value a (field `bisimTtr`).
- In addition we have the same conditions, but with the processes interchanged (the field names are the same as before but with an “r” added).

The definition of DRW-bisimilarity in case of `Process+` in CSP-Agda is as follows:

```
record Bisimw+ {i : Size}{lu : LUniv}{c : Choice}
```

($P \ P' : \text{Process}+ \infty \{lu\} \ c) : \text{Set}$ where

coinductive
field

$\text{bisimdiv} : \text{DivergentProcess}+ \ i \ c \ P \rightarrow \text{DivergentProcess}+ \ i \ c \ P'$
 $\text{nondiv}+ : \text{NonDivergent}+ \ \{i\} \ P \rightarrow \text{NonDivergent}+ \ \{i\} \ P'$
 $\text{bisimEP}' : (e : \text{ChoiceSet} \ (\text{E} \ P)) \rightarrow \text{Process}\infty \ \infty \ \{lu\} \ c$
 $\text{bisimEtr} : (e : \text{ChoiceSet} \ (\text{E} \ P))$
 $\quad \rightarrow P' \rightarrow +^* [\text{Lab} \ P \ e :: []] (\text{forcep} \ (\text{bisimEP}' \ e))$
 $\text{bisimEnext} : (e : \text{ChoiceSet} \ (\text{E} \ P))$
 $\quad \rightarrow \text{Bisimw}\infty \ \{i\} \ (\text{PE} \ P \ e) \ (\text{bisimEP}' \ e)$
 $\text{bisimIP}' : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P)) \rightarrow \text{Process}\infty \ \infty \ \{lu\} \ c$
 $\text{bisimltr} : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P))$
 $\quad \rightarrow P' \rightarrow +^* [[]] (\text{forcep} \ (\text{bisimIP}' \ \text{int1}))$
 $\text{bisimlnext} : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P))$
 $\quad \rightarrow \text{Bisimw}\infty \ \{i\} \ (\text{PI} \ P \ \text{int1}) \ (\text{bisimIP}' \ \text{int1})$
 $\text{bisimTtr} : (t : \text{ChoiceSet} \ (\text{T} \ P)) \rightarrow \text{TrP}+ \ [[]] (\text{inj}_2 \ (\text{PT} \ P \ t)) \ P'$
 $\text{bisimdivr} : \text{DivergentProcess}+ \ i \ c \ P' \rightarrow \text{DivergentProcess}+ \ i \ c \ P$
 $\text{nondiv}+r : \text{NonDivergent}+ \ \{i\} \ P' \rightarrow \text{NonDivergent}+ \ \{i\} \ P$
 $\text{bisimEP}'r : (e : \text{ChoiceSet} \ (\text{E} \ P')) \rightarrow \text{Process}\infty \ \infty \ \{lu\} \ c$
 $\text{bisimEtrr} : (e : \text{ChoiceSet} \ (\text{E} \ P'))$
 $\quad \rightarrow \text{TrP}+ \ (\text{Lab} \ P' \ e :: []) (\text{inj}_1 \ (\text{forcep} \ (\text{bisimEP}'r \ e)))) \ P$
 $\text{bisimEnextr} : (e : \text{ChoiceSet} \ (\text{E} \ P'))$
 $\quad \rightarrow \text{Bisimw}\infty \ \{i\} \ (\text{bisimEP}'r \ e) \ (\text{PE} \ P' \ e)$
 $\text{bisimIP}'r : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P')) \rightarrow \text{Process}\infty \ \infty \ \{lu\} \ c$
 $\text{bisimltrr} : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P'))$
 $\quad \rightarrow \text{TrP}+ \ [[]] (\text{inj}_1 \ (\text{forcep} \ (\text{bisimIP}'r \ \text{int1})))) \ P$
 $\text{bisimlnextr} : (\text{int1} : \text{ChoiceSet} \ (\text{I} \ P'))$
 $\quad \rightarrow \text{Bisimw}\infty \ \{i\} \ (\text{bisimIP}'r \ \text{int1}) \ (\text{PI} \ P' \ \text{int1})$
 $\text{bisimTtrr} : (t : \text{ChoiceSet} \ (\text{T} \ P')) \rightarrow \text{TrP}+ \ [[]] (\text{inj}_2 \ (\text{PT} \ P' \ t)) \ P$

Here $\rightarrow +^* []$ stands for the following:

$\rightarrow +^* [] : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} \ c)$
 $\quad (l : \text{List} \ (\text{Label} \ lu))$
 $\quad (Q : \text{Process} \ \infty \ \{lu\} \ c) \rightarrow \text{Set}$
 $\rightarrow +^* [] \ \{lu\} \ \{c\} \ P \ l \ Q = \text{TrP}+ \ \{lu\} \ \{c\} \ l \ (\text{inj}_1 \ Q) \ P$

Finally bisimilarity for $\text{Process}\infty$ is bisimilarity of the underlying Process :

$\text{record Bisimw}\infty \ \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\}$
 $\quad (P \ P' : \text{Process}\infty \ \infty \ \{lu\} \ c) : \text{Set}$ where

coinductive
field

$\text{forceB} : \{j : \text{Size} < i\} \rightarrow \text{Bisimw } \{j\}\{lu\} (\text{forcep } P) (\text{forcep } P')$

10.3 Proof of Reflexivity for Strong Bisimilarity

Proof of reflexivity for strong bisimilarity is straightforward:

Theorem 10.3.1 (Agda Theorem)

$\text{BismsRef}_\infty : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process}_\infty \infty \{lu\} c)$
 $\rightarrow \text{Bisms}_\infty \{i\} P P$
 $\text{BismsRef} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c)$
 $\rightarrow \text{Bisms} \{i\} P P$
 $\text{BismsRef}_+ : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process}_+ \infty \{lu\} c)$
 $\rightarrow \text{Bisms}_+ \{i\} P P$

Proof:

$\text{forceB } (\text{BismsRef}_\infty P) = \text{BismsRef } (\text{forcep } P)$

$\text{BismsRef } (\text{terminate } x) = \text{eqterminate}$
 $\text{BismsRef } (\text{node } P) = \text{eqnode } (\text{BismsRef}_+ P)$

$\text{bisim2E } (\text{BismsRef}_+ P) e = e$
 $\text{bisimELab } (\text{BismsRef}_+ P) e = \text{refl}$
 $\text{bisimENext } (\text{BismsRef}_+ P) e = \text{BismsRef}_\infty (\text{PE } P e)$
 $\text{bisim2I } (\text{BismsRef}_+ P) e = e$
 $\text{bisimINext } (\text{BismsRef}_+ P) e = \text{BismsRef}_\infty (\text{PI } P e)$
 $\text{bisim2T } (\text{BismsRef}_+ P) e = e$
 $\text{bisim2TEq } (\text{BismsRef}_+ P) e = \text{refl}$
 $\text{bisim2Er } (\text{BismsRef}_+ P) e = e$
 $\text{bisimELabr } (\text{BismsRef}_+ P) e = \text{refl}$

```

bisimENextr (BismsRef+ P) e = BismsRef∞ (PE P e)
bisim2lr    (BismsRef+ P) e = e
bisimlNextr (BismsRef+ P) e = BismsRef∞ (PI P e)
bisim2Tr    (BismsRef+ P) e = e
bisim2TEqr  (BismsRef+ P) e = refl

```

10.4 Proof of Reflexivity for Divergence-Respecting Weak Bisimilarity

A proof that divergence-respecting weak bisimilarity is reflexive, is again straightforward:

Theorem 10.4.1 (Agda Theorem)

```

BismwRef∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  → Bismw∞ {i} P P
BismwRef   : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  → Bismw {i} P P
BismwRef+  : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  → Bismw+ {i} P P

```

Proof:

```

forceB (BismwRef∞ {i} {lu} P) {j} = BismwRef {j} {lu} (forceP P)

```

```

BismwRef (terminate x) = eqterminate termeqterm
BismwRef (node x)      = eqnode (BismwRef+ x)

```

```

bisimdiv    (BismwRef+ P) e = e
nondiv+     (BismwRef+ P) e = e
bisimEP'    (BismwRef+ P) e = PE P e
bisimEtr    (BismwRef+ P) e =
  extc [] (inj1 (forceP (PE P e))) e (reflTrP∞ (PE P e))
bisimENext  (BismwRef+ P) e = BismwRef∞ (PE P e)

```

```

bisimIP'   (BismwRef+ P) i = PI P i
bisimltr   (BismwRef+ P) e =
  intc [] (inj1 (forcep (PI P e))) e (reflTrP∞ (PI P e))
bisimInext (BismwRef+ P) e = BismwRef∞ (PI P e)
bisimTtr   (BismwRef+ P) e = terc e
bisimdivr  (BismwRef+ P) e = e
nondiv+r   (BismwRef+ P) e = e
bisimEP'r  (BismwRef+ P) e = PE P e
bisimEtrr  (BismwRef+ P) e =
  extc [] (inj1 (forcep (PE P e))) e (reflTrP∞ (PE P e))
bisimEnextr (BismwRef+ P) e = BismwRef∞ (PE P e)
bisimIP'r  (BismwRef+ P) i = PI P i
bisimltrr  (BismwRef+ P) e =
  intc [] (inj1 (forcep (PI P e))) e (reflTrP∞ (PI P e))
bisimInextr (BismwRef+ P) e = BismwRef∞ (PI P e)
bisimTtrr  (BismwRef+ P) e = terc e

```

10.5 Proof of Symmetry for Divergence-Respecting Weak Bisimilarity

Note that in case of strong bisimilarity, the definition 10.1 is symmetric. So, if R is a bisimilarity, then R^{-1} is. Therefore we obtain a straightforward proof that divergence-respecting weak bisimilarity is symmetric:

Theorem 10.5.1 (Agda Theorem)

```

BismwSym∞ : {i : Size}{lu : LUniv}
             {c : Choice}
             (P P' : Process∞ ∞ {lu} c)
             (PP' : Bismw∞ {i} P P')
             → Bismw∞ {i} P' P

```

```

BismwSym   : {i : Size}{lu : LUniv}{c : Choice}
             (P P' : Process ∞ {lu} c)
             (PP' : Bismw {i} P P')
             → Bismw {i} P' P

```

```

BismwSym+  : {i : Size}{lu : LUniv}{c : Choice}
             (P P' : Process+ ∞ {lu} c)
             (PP' : Bismw+ {i} P P')

```

$$\rightarrow \text{Bisimw}+ \{i\} P' P$$

Proof:

$$\text{forceB} (\text{BismwSym}\infty P P' PP') = \text{BismwSym} (\text{forceP} P) (\text{forceP} P') (\text{forceB} PP')$$

$$\text{BismwSym} (\text{terminate } x) (\text{terminate } .x) (\text{eqterminate termeqterm}) = \text{eqterminate termeqterm}$$

$$\text{BismwSym} (\text{terminate } x) (\text{terminate } .x) (\text{eqterminater termeqterm}) = \text{eqterminater termeqterm}$$

$$\text{BismwSym} (\text{terminate } x) (\text{node } P') (\text{eqterminate} (\text{termeqnode } terequivP)) = \text{eqterminater} (\text{termeqnode } terequivP)$$

$$\text{BismwSym} (\text{node } P) (\text{terminate } x) (\text{eqterminater} (\text{termeqnode } terequivP)) = \text{eqterminate} (\text{termeqnode } terequivP)$$

$$\text{BismwSym} (\text{node } P) (\text{node } P') (\text{eqnode } PP') = \text{eqnode} (\text{BismwSym}+ P P' PP')$$

$$\text{bisimdiv} (\text{BismwSym}+ P P' PP') = \text{bisimdivr } PP'$$

$$\text{nondiv}+ (\text{BismwSym}+ P P' PP') = \text{nondiv}+r PP'$$

$$\text{bisimEP}' (\text{BismwSym}+ P P' PP') = \text{bisimEP}'r PP'$$

$$\text{bisimEtr} (\text{BismwSym}+ P P' PP') = \text{bisimEtrr } PP'$$

$$\text{bisimEnext} (\text{BismwSym}+ P P' PP') e = \text{BismwSym}\infty (\text{bisimEP}'r PP' e) (\text{PE } P' e) (\text{bisimEnextr } PP' e)$$

$$\text{bisimIP}' (\text{BismwSym}+ P P' PP') = \text{bisimIP}'r PP'$$

$$\text{bisimltr} (\text{BismwSym}+ P P' PP') = \text{bisimltrr } PP'$$

$$\text{bisimlnext} (\text{BismwSym}+ P P' PP') e = \text{BismwSym}\infty (\text{bisimIP}'r PP' e) (\text{PI } P' e) (\text{bisimlnextr } PP' e)$$

$$\text{bisimTtr} (\text{BismwSym}+ P P' PP') = \text{bisimTtrr } PP'$$

$$\text{bisimdivr} (\text{BismwSym}+ P P' PP') = \text{bisimdiv } PP'$$

$$\text{nondiv}+r (\text{BismwSym}+ P P' PP') = \text{nondiv}+ PP'$$

$$\text{bisimEP}'r (\text{BismwSym}+ P P' PP') = \text{bisimEP}' PP'$$

$$\text{bisimEtrr} (\text{BismwSym}+ P P' PP') = \text{bisimEtr } PP'$$

$$\text{bisimEnextr} (\text{BismwSym}+ P P' PP') e = \text{BismwSym}\infty (\text{PE } P e) (\text{bisimEP}' PP' e) (\text{bisimEnext } PP' e)$$

$$\text{bisimIP}'r (\text{BismwSym}+ P P' PP') = \text{bisimIP}' PP'$$

$$\text{bisimltrr} (\text{BismwSym}+ P P' PP') = \text{bisimltr } PP'$$

$$\text{bisimlnextr} (\text{BismwSym}+ P P' PP') e = \text{BismwSym}\infty (\text{PI } P e) (\text{bisimIP}' PP' e) (\text{bisimlnext } PP' e)$$

$$\text{bisimTtrr} \ (\text{BismwSym} + P \ P' \ PP') = \text{bisimTtr} \ PP'$$

10.6 Proof that Strong Bisimilarity Implies Divergence-Respecting Weak Bisimilarity

It is a quite natural consequence of the definitions that every strong bisimulation is a DRW-bisimulation, and therefore strong bisimilarity implies DRW-bisimilarity. For example, it is straightforward to show that, if P and Q are strongly bisimilar, then $(P \uparrow \Leftrightarrow Q \uparrow)$. A proof that strong bisimilarity implies divergence-respecting weak bisimilarity is as follows:

Theorem 10.6.1 (Agda Theorem)

$$\begin{aligned} \text{bisimsToBismw}\infty & : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P \ P' : \text{Process}\infty \ \infty \ \{lu\} \ c) \\ & \rightarrow \text{Bisims}\infty \ \{i\} \ P \ P' \\ & \rightarrow \text{Bisimw}\infty \ \{i\} \ P \ P' \end{aligned}$$

$$\begin{aligned} \text{bisimsToBismw} & : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ & \rightarrow \text{Bisims} \ \{i\} \ P \ P' \\ & \rightarrow \text{Bisimw} \ \{i\} \ P \ P' \end{aligned}$$

$$\begin{aligned} \text{bisimsToBismw}+ & : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P \ P' : \text{Process}+ \ \infty \ \{lu\} \ c) \\ & \rightarrow \text{Bisims}+ \ \{i\} \ P \ P' \\ & \rightarrow \text{Bisimw}+ \ \{i\} \ P \ P' \end{aligned}$$

Proof:

$$\begin{aligned} \text{forceB} \ (\text{bisimsToBismw}\infty \ \{i\} \ P \ P' \ PP') \ \{j\} & = \text{bisimsToBismw} \\ & \ (\text{forceP} \ P) \ (\text{forceP} \ P') \ (\text{forceB} \ PP' \ \{j\}) \end{aligned}$$

$$\begin{aligned} \text{bisimsToBismw} \ .(\text{terminate} \ a) \ .(\text{terminate} \ a) \ (\text{eqterminate} \ \{a\}) & = \\ \text{eqterminate} \ \text{termeqterm} & \\ \text{bisimsToBismw} \ .(\text{node} \ Q) \ .(\text{node} \ Q') \ (\text{eqnode} \ \{Q\} \ \{Q'\} \ x) & = \end{aligned}$$

$\text{eqnode } (\text{bisimsToBismw+ } Q \ Q' \ x)$

$\text{bisimdiv } (\text{bisimsToBismw+ } P \ P' \ PP') = \text{divLemBisims+ } P \ P' \ PP'$
 $\text{nondiv+ } (\text{bisimsToBismw+ } P \ P' \ PP') = \text{nondivLemBisims+ } P \ P' \ PP'$
 $\text{bisimEP'} (\text{bisimsToBismw+ } P \ P' \ PP') \ e = \text{PE } P' (\text{bisim2E } PP' \ e)$
 $\text{bisimEtr } (\text{bisimsToBismw+ } P \ P' \ PP') \ e \text{ rewrite } (\text{bisimELab } PP' \ e) =$
 $\quad \text{extc [] } (\text{inj}_1 (\text{forcep } (\text{PE } P' (\text{bisim2E } PP' \ e))))$
 $\quad (\text{bisim2E } PP' \ e) (\text{reflTrP}\infty (\text{PE } P' (\text{bisim2E } PP' \ e)))$
 $\text{bisimEnext } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{bisimsToBismw}\infty (\text{PE } P \ e) (\text{PE } P' (\text{bisim2E } PP' \ e))$
 $\quad (\text{bisimENext } PP' \ e)$
 $\text{bisimIP'} (\text{bisimsToBismw+ } P \ P' \ PP') \ e = \text{PI } P' (\text{bisim2I } PP' \ e)$
 $\text{bisimltr } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{intc [] } ((\text{inj}_1 (\text{forcep } (\text{PI } P' (\text{bisim2I } PP' \ e)))))$
 $\quad (\text{bisim2I } PP' \ e) (\text{reflTrP}\infty ((\text{PI } P' (\text{bisim2I } PP' \ e))))$
 $\text{bisimlnext } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{bisimsToBismw}\infty (\text{PI } P \ e) (\text{PI } P' (\text{bisim2I } PP' \ e))$
 $\quad (\text{bisimlnext } PP' \ e)$
 $\text{bisimTtr } (\text{bisimsToBismw+ } P \ P' \ PP') \ t \text{ rewrite } (\text{bisim2TEq } PP' \ t) =$
 $\quad \text{terc } (\text{bisim2T } PP' \ t)$
 $\text{bisimdivr } (\text{bisimsToBismw+ } P \ P' \ PP') = \text{divLemBisims+r } P \ P' \ PP'$
 $\text{nondiv+r } (\text{bisimsToBismw+ } P \ P' \ PP') = \text{nondivLemBisims+r } P \ P' \ PP'$
 $\text{bisimEP'r } (\text{bisimsToBismw+ } P \ P' \ PP') \ e = \text{PE } P (\text{bisim2Er } PP' \ e)$
 $\text{bisimEtrr } (\text{bisimsToBismw+ } P \ P' \ PP') \ e \text{ rewrite } (\text{bisimELabr } PP' \ e) =$
 $\quad \text{extc [] } (\text{inj}_1 (\text{forcep } (\text{PE } P (\text{bisim2Er } PP' \ e)))))$
 $\quad (\text{bisim2Er } PP' \ e) (\text{reflTrP}\infty (\text{PE } P (\text{bisim2Er } PP' \ e)))$
 $\text{bisimEnext r } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{bisimsToBismw}\infty (\text{PE } P (\text{bisim2Er } PP' \ e)) (\text{PE } P' \ e)$
 $\quad (\text{bisimENextr } PP' \ e)$
 $\text{bisimIP'r } (\text{bisimsToBismw+ } P \ P' \ PP') \ e = \text{PI } P (\text{bisim2lr } PP' \ e)$
 $\text{bisimltrr } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{intc [] } (\text{inj}_1 (\text{forcep } (\text{PI } P (\text{bisim2lr } PP' \ e)))))$
 $\quad (\text{bisim2lr } PP' \ e) (\text{reflTrP}\infty (\text{PI } P (\text{bisim2lr } PP' \ e)))$
 $\text{bisimlnextr } (\text{bisimsToBismw+ } P \ P' \ PP') \ e =$
 $\quad \text{bisimsToBismw}\infty (\text{PI } P (\text{bisim2lr } PP' \ e)) (\text{PI } P' \ e)$
 $\quad ((\text{bisimlnextr } PP' \ e))$
 $\text{bisimTtrr } (\text{bisimsToBismw+ } P \ P' \ PP') \ t \text{ rewrite } (\text{bisim2TEqr } PP' \ t) =$
 $\quad \text{terc } (\text{bisim2Tr } PP' \ t)$

10.7 Bisimilarity Implies Trace Equivalence

As we have seen before, weak bisimilarity can not distinguish between the primitive processes like STOP and DIV. In fact weak bisimilarity does not imply any CSP semantic equality other than trace equivalence. However, divergence-respecting weak bisimilarity respect all such models: if two LTS nodes are divergence-respecting weakly bisimilar, they are equivalent in all standard CSP semantics. Here we prove that strong and divergence-respecting weak bisimilarity imply trace equivalence.

10.7.1 Strong Bisimilarity Implies Trace Equivalence

We give here a proof that strong bisimilarity implies trace equivalence. Since the proof that strong bisimilarity implies trace equivalence is more straightforward and natural, we give the direct proof here.

The proof is obtained by taking a trace for one process and replacing each step by steps in the other process. The proof that strong bisimilarity implies refinement with respect to traces is as follows:

$$\text{SbisimTraceEq}_\infty : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process}_\infty \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisims}_\infty \ \{\infty\} \ P \ P') \rightarrow P \sqsubseteq_\infty P'$$

$$\text{SbisimTraceEq}_\infty \ \{lu\}\{c\} \ P \ P' \ PP' \ l \ m \ tr = \text{SbisimTraceEq} \\ (\text{forcep} \ P) \ (\text{forcep} \ P') \ (\text{forceB} \ PP') \ l \ m \ tr$$

$$\text{SbisimTraceEq} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process} \ \infty \ c) \\ (PP' : \text{Bisims} \ \{\infty\} \ P \ P') \rightarrow P \sqsubseteq P'$$

$$\text{SbisimTraceEq} \ .(\text{terminate} \ x) \ .(\text{terminate} \ x) \ \text{eqterminate} \\ .[] \ .(\text{just} \ x) \ (\text{ter} \ x) = \text{ter} \ x$$

$$\text{SbisimTraceEq} \ .(\text{terminate} \ x) \ .(\text{terminate} \ x) \ \text{eqterminate} \\ .[] \ .\text{nothing} \ (\text{empty} \ x) = \text{empty} \ x$$

$$\text{SbisimTraceEq} \ .(\text{node} \ P) \ .(\text{node} \ P') \ (\text{eqnode} \ \{P\} \ \{P'\} \ PP') \\ .[] \ .\text{nothing} \ (\text{tnode} \ \text{empty}) = \text{tnode} \ \text{empty}$$

$$\text{SbisimTraceEq} \ \{lu\}\{c\} \ .(\text{node} \ P) \ .(\text{node} \ P') \ (\text{eqnode} \ \{P\} \ \{P'\} \ PP') \\ .(\text{Lab} \ P' \ x :: l) \ mc \ (\text{tnode} \ (\text{extc} \ l \ .mc \ x \ tr_2')) = \text{tnode} \ tr$$

where

$$Q' : \text{Process}_\infty \ \infty \ \{lu\} \ c \\ Q' = \text{PE} \ P' \ x$$

$Q : \text{Process} \infty \infty \{lu\} c$
 $Q = \text{bisimEP}'r \text{ (bisimsToBismw+ } P P' PP') x$

$QQ' : \text{Bisimw} \infty Q Q'$
 $QQ' = \text{bisimEnextr} \text{ (bisimsToBismw+ } P P' PP') x$

$tr_1 : P \rightarrow +^* [\text{Lab } P' x :: []] (\text{forcep } Q)$
 $tr_1 = \text{bisimEtrr} \text{ (bisimsToBismw+ } P P' PP') x$

$tr_2'' : \text{Tr} \infty \{lu\}\{c\} l mc Q'$
 $tr_2'' = tr_2'$

$tr_2 : \text{Tr} \infty \{lu\}\{c\} l mc Q$
 $tr_2 = \text{bisimTraceEq} \infty \{lu\}\{c\} Q Q' QQ' l mc tr_2''$

$tr : \text{Tr+ } \{lu\}\{c\} (\text{Lab } P' x :: l) mc P$
 $tr = \text{traceAppendTrw+ } c P (\text{forcep } Q) (\text{Lab } P' x :: []) l mc tr_1 tr_2$

$\text{SbisimTraceEq } \{lu\}\{c\} .(\text{node } P) .(\text{node } P') (\text{eqnode } \{P\} \{P'\} PP')$
 $l mc (\text{tnode } (\text{intc } l . mc x tr_2')) = \text{tnode } tr$

where

$Q' : \text{Process} \infty \infty \{lu\} c$
 $Q' = \text{PI } P' x$

$Q : \text{Process} \infty \infty \{lu\} c$
 $Q = \text{bisimIP}'r \text{ (bisimsToBismw+ } P P' PP') x$

$QQ' : \text{Bisimw} \infty Q Q'$
 $QQ' = \text{bisimInextr} \text{ (bisimsToBismw+ } P P' PP') x$

$tr_1 : P \rightarrow +^* [[]] (\text{forcep } Q)$
 $tr_1 = \text{bisimltrr} \text{ (bisimsToBismw+ } P P' PP') x$

$tr_2'' : \text{Tr} \infty \{lu\}\{c\} l mc Q'$
 $tr_2'' = tr_2'$

$tr_2 : \text{Tr} \infty \{lu\}\{c\} l mc Q$
 $tr_2 = \text{bisimTraceEq} \infty \{lu\}\{c\} Q Q' QQ' l mc tr_2''$

$tr : \text{Tr+ } \{lu\}\{c\} ([[] ++ l) mc P$
 $tr = \text{traceAppendTrw+ } c P (\text{forcep } Q) [] l mc tr_1 tr_2$

```

SbisimTraceEq .(node P) .(node P') (eqnode {P} {P'} PP')
               .[] .(just (PT P' x)) (tnode (terc x)) =
               tnode (trPtoTr+ [] (inj2 (PT P' x)) P tr1)

where
  tr1      : TrP+ [] (inj2 (PT P' x)) P
  tr1      =  bisimTtrr (bisimsToBismw+ P P' PP') x

```

10.7.2 Divergence-Respecting Weak Bisimilarity Implies Trace Equivalence

The proof, that divergence-respecting weak bisimilarity implies trace equivalence is similar to the one before, and the first cases are straightforward. In case of external and internal choice, we have to define the trace by combining the traces from process P to subprocess Q with the trace from a process Q to the maybe terminated process mc . A proof that *divergence-respecting* weak bisimilarity implies trace equivalence is as follows:

Theorem 10.7.1 (Agda Theorem)

```

bisimTraceEq∞ : {lu : LUniv}{c : Choice}
                (P P' : Process∞ ∞ {lu} c)
                (PP' : Bisimw∞ {∞} P P')
                → P ⊆∞ P'

```

```

bisimTraceEq   : {lu : LUniv}{c : Choice}
                (P P' : Process ∞ c)
                (PP' : Bisimw {∞} P P')
                → P ⊆ P'

```

```

bisimTraceEq+  : {lu : LUniv}{c : Choice}
                (P P' : Process+ ∞ c)
                (PP' : Bisimw+ {∞} P P')
                → P ⊆+ P'

```

Proof:

```

bisimTraceEq∞ P P' PP' l m tr = bisimTraceEq (forcep P)
                                   (forcep P') (forceB PP' {∞}) l m tr

```

```

bisimTraceEq  .(terminate x) .(terminate x) (eqterminate termeqterm)
               .[] .(just x) (ter x) = ter x
bisimTraceEq P .(terminate x) (eqterminater terequivP)
               .[] .(just x) (ter x) =
               termEquivalentImpliesTrace P terequivP
bisimTraceEq  .(terminate x) .(terminate x) (eqterminate termeqterm)
               .[] .nothing (empty x) = empty x
bisimTraceEq P .(terminate x) (eqterminater terequivP)
               .[] .nothing (empty x) =
               termEquivalentImpliesTraceEmpty P terequivP
bisimTraceEq  .(terminate a) .(node P) (eqterminate {a} terEquivP)
               .l x (tnode {l} {x} {P} tr) =
               termEquivalentTracelsTerTrace+ P l terEquivP tr
bisimTraceEq  .(node Q) .(node Q') (eqnode {Q} {Q'} QQ').[] .nothing
               (tnode {[]} {nothing} {Q} empty) = tnode empty
bisimTraceEq {lu}{c} .(node P) .(node P') (eqnode {P} {P'} PP')
               .(Lab P' x :: l) .mc (tnode (extc l mc x tr₂')) = tnode tr
where
  Q' : Process $\infty$   $\infty$  {lu} c
  Q' = PE P' x

  Q : Process $\infty$   $\infty$  {lu} c
  Q = bisimEP'r PP' x

  QQ' : Bisimw $\infty$  Q Q'
  QQ' = bisimEnextr PP' x

  tr₁ : P  $\rightarrow$  +*[ Lab P' x :: [] ] (forcep Q)
  tr₁ = bisimEtrr PP' x

  tr₂'' : Tr $\infty$  {lu} {c} l mc Q'
  tr₂'' = tr₂'

  tr₂ : Tr $\infty$  {lu} {c} l mc Q
  tr₂ = bisimTraceEq $\infty$  {lu} {c} Q Q' QQ' l mc tr₂''

  tr : Tr+ {lu} {c} (Lab P' x :: l) mc P
  tr = traceAppendTrw+ c P (forcep Q) (Lab P' x :: [])
                        l mc tr₁ tr₂

```

```

bisimTraceEq {lu} {c} .(node P) .(node P') (eqnode {P} {P'} PP')
  l mc (tnode (intc .l .mc x tr2')) = tnode tr

```

where

```

Q' : Process $\infty$   $\infty$  {lu} c
Q' = PI P' x

```

```

Q : Process $\infty$   $\infty$  {lu} c
Q = bisimIP'r PP' x

```

```

QQ' : Bisimw $\infty$  Q Q'
QQ' = bisimInextr PP' x

```

```

tr1 : P  $\rightarrow$ +*[ [] ] (forcep Q)
tr1 = bisimltrr PP' x

```

```

tr2 : Tr $\infty$  {lu} {c} l mc Q
tr2 = bisimTraceEq $\infty$  {lu}{c} Q Q' QQ' l mc tr2'

```

```

tr : Tr+ {lu}{c} ([[] ++ l) mc P
tr = traceAppendTrw+ c P (forcep Q) [[] l mc tr1 tr2

```

```

bisimTraceEq {lu}{c} .(node P) .(node P') (eqnode {P} {P'} PP')
  .[] .(just (PT P' x)) (tnode {[]} {.(just (PT P' x))}
    {P'} (terc x)) =
    tnode (trPtoTr+ [] (inj2 (PT P' x)) P tr1)

```

where

```

tr1 : TrP+ [] (inj2 (PT P' x)) P
tr1 = bisimTtrr PP' x

```

```

bisimTraceEq+ P P' PP' .[] .nothing empty = empty
bisimTraceEq+ {lu}{c} P P' PP' .(Lab P' x :: l1) m (extc l1 .m x tr2') = tr

```

where

```

Q' : Process $\infty$   $\infty$  {lu} c
Q' = PE P' x

```

```

Q : Process $\infty$   $\infty$  {lu} c
Q = bisimEP'r PP' x

```

$$QQ' : \text{Bisimw}_\infty Q Q'$$

$$QQ' = \text{bisimEnextr } PP' x$$

$$\text{tr}_1 : P \rightarrow +^* [\text{Lab } P' x :: []] (\text{forcep } Q)$$

$$\text{tr}_1 = \text{bisimEtrr } PP' x$$

$$\text{tr}_2'' : \text{Tr}_\infty \{lu\}\{c\} l_1 m Q'$$

$$\text{tr}_2'' = \text{tr}_2'$$

$$\text{tr}_2 : \text{Tr}_\infty \{lu\}\{c\} l_1 m Q$$

$$\text{tr}_2 = \text{bisimTraceEq}_\infty \{lu\}\{c\} Q Q' QQ' l_1 m \text{tr}_2''$$

$$\text{tr} : \text{Tr}_+ \{lu\}\{c\} (\text{Lab } P' x :: l_1) m P$$

$$\text{tr} = \text{traceAppendTrw}_+ c P (\text{forcep } Q) \\ (\text{Lab } P' x :: []) l_1 m \text{tr}_1 \text{tr}_2$$

$$\text{bisimTraceEq}_+ \{lu\}\{c\} P P' PP' l m (\text{intc } .l .m x \text{tr}_2') = \text{tr}$$

where

$$Q' : \text{Process}_\infty \infty \{lu\} c$$

$$Q' = \text{PI } P' x$$

$$Q : \text{Process}_\infty \infty \{lu\} c$$

$$Q = \text{bisimIP'r } PP' x$$

$$QQ' : \text{Bisimw}_\infty Q Q'$$

$$QQ' = \text{bisimInextr } PP' x$$

$$\text{tr}_1 : P \rightarrow +^* [[]] (\text{forcep } Q)$$

$$\text{tr}_1 = \text{bisimltrr } PP' x$$

$$\text{tr}_2 : \text{Tr}_\infty \{lu\}\{c\} l m Q$$

$$\text{tr}_2 = \text{bisimTraceEq}_\infty Q Q' QQ' l m \text{tr}_2'$$

$$\text{tr} : \text{Tr}_+ \{lu\}\{c\} ([[] ++ l] m P$$

$$\text{tr} = \text{traceAppendTrw}_+ c P (\text{forcep } Q) [[] l m \text{tr}_1 \text{tr}_2$$

$$\text{bisimTraceEq}_+ P P' PP' . [[]] . (\text{just } (\text{PT } P' t)) (\text{terc } t) = \\ \text{trPtoTr}_+ [[]] (\text{inj}_2 (\text{PT } P' t)) P \text{tr}_1$$

where

$$\text{tr}_1 : \text{TrP}_+ [[]] (\text{inj}_2 (\text{PT } P' t)) P$$

$$\text{tr}_1 = \text{bisimTtrr } PP' t$$

We obtain by symmetry of DRW-bisimulation as well refinement in the other direction:

$$\begin{aligned} \text{bisimTraceEq}\infty r &: \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process}\infty \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw}\infty \ \{\infty\} \ P \ P') \rightarrow P' \sqsubseteq_{\infty} P \\ \text{bisimTraceEq}\infty r \ P \ P' \ PP' &= \text{bisimTraceEq}\infty \ P' \ P \ (\text{BismwSym}\infty \ P \ P' \ PP') \end{aligned}$$

$$\begin{aligned} \text{bisimTraceEq}+r &: \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process}+ \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw}+ \ \{\infty\} \ P \ P') \rightarrow P' \sqsubseteq_+ P \\ \text{bisimTraceEq}+r \ P \ P' \ PP' &= \text{bisimTraceEq}+ \ P' \ P \ (\text{BismwSym}+ \ P \ P' \ PP') \end{aligned}$$

$$\begin{aligned} \text{bisimTraceEq}r &: \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \rightarrow P' \sqsubseteq P \\ \text{bisimTraceEq}r \ P \ P' \ PP' &= \text{bisimTraceEq} \ P' \ P \ (\text{BismwSym} \ P \ P' \ PP') \end{aligned}$$

This implies that DRW-bisimulation implies trace equivalence equivalence:

Theorem 10.7.2 (Agda Theorem)

$$\begin{aligned} \text{bisimTraceEq}\infty = &: \{lu : \text{LUniv}\}\{c : \text{Choice}\} \\ &\quad (P \ P' : \text{Process}\infty \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw}\infty \ \{\infty\} \ P \ P') \\ &\rightarrow P \equiv_{\infty} P' \end{aligned}$$

$$\begin{aligned} \text{bisimTraceEq}+ = &: \{lu : \text{LUniv}\}\{c : \text{Choice}\} \\ &\quad (P \ P' : \text{Process}+ \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw}+ \ \{\infty\} \ P \ P') \\ &\rightarrow P \equiv_+ P' \end{aligned}$$

$$\begin{aligned} \text{bisimTraceEq} = &: \{lu : \text{LUniv}\}\{c : \text{Choice}\} \\ &\quad (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\ &\rightarrow P \equiv P' \end{aligned}$$

Proof:

$$\begin{aligned} \text{bisimTraceEq}\infty = \ P \ P' \ PP' &= \text{bisimTraceEq}\infty \ P \ P' \ PP' , \\ &\quad \text{bisimTraceEq}\infty r \ P \ P' \ PP' \end{aligned}$$

$$\text{bisimTraceEq} += P P' PP' = \text{bisimTraceEq} + P P' PP', \\ \text{bisimTraceEq} + P P' PP'$$

$$\text{bisimTraceEq} = P P' PP' = \text{bisimTraceEq} P P' PP', \\ \text{bisimTraceEq} P P' PP'$$

We can get now an alternative proof that strong bisimilarity implies trace equivalence:

Theorem 10.7.3 (Agda Theorem)

$$\text{bisimTraceEqs} \infty = : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P P' : \text{Process} \infty \infty \{lu\} c) \\ (PP' : \text{Bisims} \infty \{\infty\} P P') \\ \rightarrow P \equiv \infty P'$$

$$\text{bisimTraceEqs} += : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P P' : \text{Process} + \infty \{lu\} c) \\ (PP' : \text{Bisims} + \{\infty\} P P') \\ \rightarrow P \equiv + P'$$

$$\text{bisimTraceEqs} = : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P P' : \text{Process} \infty \{lu\} c) \\ (PP' : \text{Bisims} \{\infty\} P P') \\ \rightarrow P \equiv P'$$

Proof:

$$\text{bisimTraceEqs} \infty = P P' PP' = \\ \text{bisimTraceEq} \infty = P P' (\text{bisimsToBismw} \infty P P' PP')$$

$$\text{bisimTraceEqs} += P P' PP' = \\ \text{bisimTraceEq} += P P' (\text{bisimsToBismw} + P P' PP')$$

$$\text{bisimTraceEqs} = P P' PP' = \\ \text{bisimTraceEq} = P P' (\text{bisimsToBismw} P P' PP')$$

10.8 Proof of Lemma 2.4.6 Part 1

In Sect. 2.4.5 we proved the Key Lemma for DRW bisimulation (Lemma 2.4.6) as taken from Roscoe [2010, 1998]. This lemma shows that DRW-bisimilarity respects stable states. In this section, we will go through the proof using standard CSP, and then prove it step by step in Agda. We repeat Lemma 2.4.6, but with the bisimilarity R replaced by \sim , since we will later use R denote a process:

Lemma (*repetition of Lemma 2.4.6*) *Let \sim be a DRW-bisimulation.*

- $\forall P, P', Q \in S'. \forall s \in \Sigma^{*, \checkmark}. P \sim P' \wedge P \xrightarrow{s} Q$
 $\Rightarrow \exists Q' \in S'. P' \xrightarrow{s} Q' \wedge Q \sim Q' \wedge (\text{stable}(Q) \Rightarrow \text{stable}(Q'))$
- $\forall P, P', Q' \in S'. \forall x \in \Sigma^{*, \checkmark}. P \sim P' \wedge P' \xrightarrow{s} Q'$
 $\Rightarrow \exists Q \in S'. P \xrightarrow{s} Q \wedge Q \sim Q' \wedge (\text{stable}(Q') \Rightarrow \text{stable}(Q))$

In chapter 2 we proved the first direction for normal CSP. In this section, in order to motivate the Agda proof, repeat this proof step by step, but for the second direction, and show how each step is proved in Agda.

Assume we have the following:

$$P \sim P' \wedge P' \xrightarrow{s} Q' \wedge \text{stable}(Q') \quad (*)$$

Then by \sim being a DRW-bisimulation we obtain a Q such that

$$Q \sim Q' \wedge P \xrightarrow{s} Q$$

By $Q \sim Q' \wedge \text{stable}(Q')$ we get Q' and therefore as well Q are non-divergent processes.

Therefore there exists \hat{Q} such that

$$Q \Rightarrow \hat{Q} \wedge \text{stable}(\hat{Q})$$

By $Q \sim Q'$ there exists \hat{Q}' , such that $Q' \Rightarrow \hat{Q}'$ and $\hat{Q} \sim \hat{Q}'$. By stability of Q' we get $Q' = \hat{Q}'$. Therefore we get $\text{stable}(\hat{Q})$ and $\hat{Q} \sim Q'$.

In case we just had in $(*)$ Q' without stability, we just get Q which is DRW-bisimilar to Q' and a trace s from P to Q .

In figure 10.1, we display the situation, except that the processes are called R, R', \dots instead of Q, Q', \dots in order to coincide with the notation used in Agda (where we work directly with bisimilarity instead of bisimulation, so there is no occurrence of the relation R). So we have two process P and P' which are bisimilar, and we have a subprocess R' and trace from P' to R'

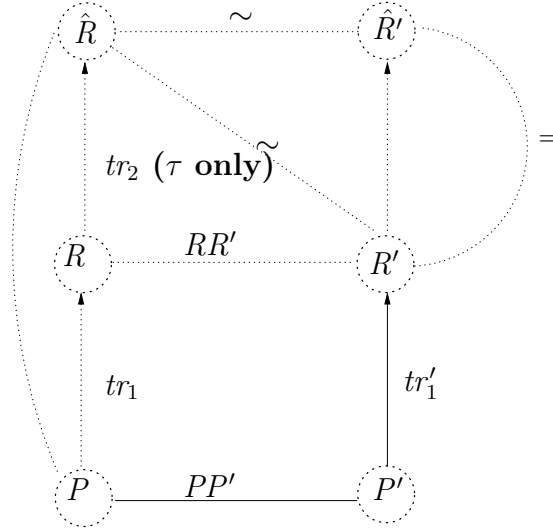


Figure 10.1: Proof of Lemma 2.4.6

where R' is stable. By bisimilarity, we can obtain a process R and a trace from P to R such that R and R' are bisimilar. The subprocess R may be stable or can do τ -events to reach \hat{R} , which is stable and bisimilar to R' .

In this Section we prove this lemma in Agda, however only obtaining Schneider instead of Roscoe stability. In Section 10.9 we will show that we obtain Roscoe stability. There we will prove as well the theorem, in case we don't assume stability Q' and therefore don't get stability for the resulting process.

In the first stage of the proof, we need to define the process R as shown in the figure 10.1, in Agda we define it as follows:

```

bisimTraceTrP $\infty$ 1 : {lu : LUniv}{c : Choice}(P P' : Process $\infty$   $\infty$  c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')(l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP $\infty$  l tick P')
  → Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
bisimTraceTrP $\infty$ 1 P P' PP' l x tr = bisimTraceTrP1 (forceP P)
  (forceP P')
  (forceB PP')
  l x tr

```

```

bisimTraceTrP1 : {lu : LUniv}{c : Choice}(P P' : Process  $\infty$  {lu} c)

```

```

      (PP' : Bisimw {∞} P P')
      (l : List (Label lu))
      (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
      (tr : TrP l tick P')
      → Process ∞ {lu} c ⊔ ChoiceSet c
bisimTraceTrP1 .(terminate y) (terminate x)(eqterminate {y} x1)
.[] .(inj2 x) (ter .x) = inj2 y
bisimTraceTrP1 P (terminate x) (eqterminater termequivP)
.[] .(inj2 x) (ter .x) =
  termequivPToTick1 x P termequivP
bisimTraceTrP1 .(terminate y) (terminate x)
  (eqterminate {y} x1) .[]
.(inj1 (terminate x))(empty .x)
= inj1 (terminate y)
bisimTraceTrP1 P (terminate x) (eqterminater termequivP) .[]
.(inj1 (terminate x)) (empty .x) =
  termequivPToTick1' x P termequivP
bisimTraceTrP1 .(terminate a) (node P')
  (eqterminate {a} node xTerEquiv)
  l (inj1 x) (tnode tr) = inj1 (terminate a)
bisimTraceTrP1 .(terminate a) (node P')
  (eqterminate {a} node xTerEquiv)
  l (inj2 y) (tnode tr) = inj2 a
bisimTraceTrP1 .(node P) (node P')
  (eqnode {P} PP') l tick
  (tnode tr) = bisimTraceTrP1+ P P' PP' l tick tr

```

```

bisimTraceTrP1+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ c)
  (PP' : Bisimw+ {∞} P P')
  (l : List (Label lu))
  (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  (tr : TrP+ l tick P')
  → Process ∞ {lu} c ⊔ ChoiceSet c
bisimTraceTrP1+ P P' PP' .[] .(inj1 (node P')) empty =
  (inj1 (node P))
bisimTraceTrP1+ {lu}{c} P P' PP'
. (Lab P' x :: l) tick (extc l .tick x tr) =
  bisimTraceTrP∞1 R R' RR' l tick tr

```

module bisimETraceTrP₁+auxmodule where

```

R' : Process $\infty$   $\infty$  c
R' = PE P' x

R : Process $\infty$   $\infty$  c
R = bisimEP'r PP' x

RR' : Bisimw $\infty$  { $\infty$ } R R'
RR' = bisimEnextr PP' x

bisimTraceTrP1+ {lu}{c} P P' PP' l tick
  (intc .l .tick x x1) =
    bisimTraceTrP $\infty$ 1 R R' RR' l tick x1
module bisimTraceTrP1+auxmodule where
  R' : Process $\infty$   $\infty$  c
  R' = PI P' x

  R : Process $\infty$   $\infty$  c
  R = bisimIP'r PP' x

  RR' : Bisimw $\infty$  { $\infty$ } R R'
  RR' = bisimInextr PP' x

bisimTraceTrP1+ P P' PP' .[] .(inj2 (PT P' x))
  (terc x) = inj2 (PT P' x)

```

Then we have to define the trace connecting the process P with the process R as shown in the previous figure, and this is defined in CSP-Agda as follows:

```

bisimTraceTrP $\infty$ 2 : {lu : LUniv}{c : Choice}(P P' : Process $\infty$   $\infty$  c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')(l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP $\infty$  l tick P')
  → TrP $\infty$  l (bisimTraceTrP $\infty$ 1 P P' PP' l tick tr) P
bisimTraceTrP $\infty$ 2 P P' PP' l x tr = bisimTraceTrP2 (forcep P)
  (forcep P')
  (forceB PP')
  l x tr

```

```

bisimTraceTrP2 : {lu : LUniv}{c : Choice}

```

$$\begin{aligned}
& (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\
& (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\
& (l : \text{List} \ (\text{Label} \ lu)) \\
& (tick : \text{Process} \ \infty \ \{lu\} \ c \uplus \text{ChoiceSet} \ c) \\
& (tr : \text{TrP} \ l \ tick \ P') \\
& \rightarrow \text{TrP} \ l \ (\text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \ tick \ tr) \ P \\
& \text{bisimTraceTrP}_2 \ .(\text{terminate} \ y) \ .(\text{terminate} \ x) \\
& \quad (\text{eqterminate} \ \{y\} \ x_1) \\
& \quad .[] \ .(\text{inj}_2 \ x) \ (\text{ter} \ x) = \text{ter} \ y \\
& \text{bisimTraceTrP}_2 \ P \ .(\text{terminate} \ x) \\
& \quad (\text{eqterminater} \ \text{termequiv}P) \\
& \quad .[] \ .(\text{inj}_2 \ x) \\
& \quad (\text{ter} \ x) = \text{termequivPToTick}_2 \ x \ P \ \text{termequiv}P \\
& \text{bisimTraceTrP}_2 \ .(\text{terminate} \ y) \ .(\text{terminate} \ x) \\
& \quad (\text{eqterminate} \ \{y\} \ x_1) \\
& \quad .[] \ .(\text{inj}_1 \ (\text{terminate} \ x))(\text{empty} \ x) = \text{empty} \ y \\
& \text{bisimTraceTrP}_2 \ P \ .(\text{terminate} \ x) \\
& \quad (\text{eqterminater} \ \text{termequiv}P) \ .[] \\
& \quad .(\text{inj}_1 \ (\text{terminate} \ x)) \ (\text{empty} \ x) = \\
& \quad \text{termequivPToTick}_2' \ x \ P \ \text{termequiv}P \\
& \text{bisimTraceTrP}_2 \ \{lu\} \ \{c\} \ .(\text{terminate} \ a) \ .(\text{node} \ P') \\
& \quad (\text{eqterminate} \ \{a\} \ \text{termequiv}P) \ l \\
& \quad (\text{inj}_1 \ x) \ (\text{tnode} \ \{.l\} \ \{.(\text{inj}_1 \ x)\} \ \{P'\} \ tr) \\
& \quad \text{rewrite} \\
& \quad (\text{lemmaTraceTerminationEquivalentEmpty}+' \\
& \quad \quad c \ l \ P' \ (\text{inj}_1 \ x) \ a \ \text{termequiv}P \ tr) = \text{empty} \ a \\
& \text{bisimTraceTrP}_2 \ \{lu\} \ \{c\} \ .(\text{terminate} \ a) \ .(\text{node} \ P') \\
& \quad (\text{eqterminate} \ \{a\} \ \text{termequiv}P) \ l \\
& \quad (\text{inj}_2 \ y) \ (\text{tnode} \ \{.l\} \ \{.(\text{inj}_2 \ y)\} \ \{P'\} \ tr) \\
& \quad \text{rewrite} \\
& \quad (\text{lemmaTraceTerminationEquivalentEmpty}+' \ c \ l \ P' \\
& \quad \quad (\text{inj}_2 \ y) \ a \ \text{termequiv}P \ tr) = \text{ter} \ a \\
& \text{bisimTraceTrP}_2 \ \{lu\} \ \{c\} \ .(\text{node} \ P) \ .(\text{node} \ P') \\
& \quad (\text{eqnode} \ \{P\} \ \{P'\} \ PP') \ l \ tick \\
& \quad (\text{tnode} \ tr) = \\
& \quad \text{tnode} \ (\text{bisimTraceTrP}_2+ \ P \ P' \ PP' \ l \ tick \ tr) \\
& \text{bisimTraceTrP}_2+ : \ \{lu : \text{LUniv}\} \ \{c : \text{Choice}\} \\
& \quad (P \ P' : \text{Process}+ \ \infty \ c) \\
& \quad (PP' : \text{Bisimw}+ \ \{\infty\} \ P \ P')
\end{aligned}$$

```

      (l : List (Label lu))
      (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
      (tr      : TrP+ l tick P')
    → TrP+ l (bisimTraceTrP1+ P P' PP' l tick tr) P
bisimTraceTrP2+ P P' PP' .[] .(inj1 (node P')) empty = empty
bisimTraceTrP2+ {lu}{c} P P' PP' .(Lab P' x :: l)
      tick (extc l .tick x tr1') = tr
module bisimETraceTrP2+auxmodule where
  R' : Process∞ ∞ c
  R' = PE P' x

  R : Process∞ ∞ c
  R = bisimEP'r PP' x

  RR' : Bisimw∞ {∞} R R'
  RR' = bisimEnextr PP' x

  Rhat : Process ∞ {lu} c ⊔ ChoiceSet c
  Rhat = bisimTraceTrP1 (forcep R)
      (forcep (PE P' x)) (forceB RR') l tick tr1'
  tr1 : P →+* [ Lab P' x :: [] ] (forcep R)
  tr1 = bisimEtrr PP' x

  tr2 : TrP∞ {lu}{c} l Rhat R
  tr2 = bisimTraceTrP2 (forcep R) (forcep (PE P' x))
      (forceB RR') l tick tr1'

  tr : TrP+ (Lab P' x :: l) Rhat P
  tr = trPAppendTrw+ c P (forcep R) (Lab P' x :: []) l
      Rhat tr1 tr2

bisimTraceTrP2+ {lu}{c} P P' PP' l
      tick (intc l .tick x tr2') = tr
module bisimITraceTrP2+auxmodule where
  R' : Process∞ ∞ c
  R' = PI P' x

  R : Process∞ ∞ c
  R = bisimIP'r PP' x

  RR' : Bisimw∞ {∞} R R'

```

$$RR' = \text{bisimInextr } PP' \ x$$

$$\begin{aligned} \text{Rhat} &: \text{Process} \infty \{lu\} \ c \uplus \text{ChoiceSet} \ c \\ \text{Rhat} &= \text{bisimTraceTrP}_1 (\text{forcep} (\text{bisimIP}'r \ PP' \ x)) \\ &\quad (\text{forcep} (\text{PI} \ P' \ x)) \\ &\quad (\text{forceB} (\text{bisimInextr} \ PP' \ x)) \ l \ \text{tick} \ tr_2' \end{aligned}$$

$$\begin{aligned} tr_1 &: P \rightarrow +^* [\] (\text{forcep} \ R) \\ tr_1 &= \text{bisimLtrr} \ PP' \ x \end{aligned}$$

$$\begin{aligned} tr_2 &: \text{TrP} \ l \ \text{Rhat} (\text{forcep} \ R) \\ tr_2 &= \text{bisimTraceTrP}_2 (\text{forcep} \ R) \\ &\quad (\text{forcep} (\text{PI} \ P' \ x)) \\ &\quad (\text{forceB} (\text{bisimInextr} \ PP' \ x)) \ l \ \text{tick} \ tr_2' \end{aligned}$$

$$\begin{aligned} tr &: \text{TrP}+ \ l \ \text{Rhat} \ P \\ tr &= \text{trPAppendTrw}+ \ c \ P \\ &\quad (\text{forcep} \ R) \ [\] \ l \ \text{Rhat} \ tr_1 \ tr_2 \end{aligned}$$

$$\begin{aligned} &\text{bisimTraceTrP}_2+ \ \{lu\}\{c\} \ P \ P' \ PP' \ .[\] \\ &\quad .(\text{inj}_2 (\text{PT} \ P' \ x)) (\text{terc} \ x) = tr_1 \\ &\text{where} \\ &\quad tr_1 : \text{TrP}+ \ [\] (\text{inj}_2 (\text{PT} \ P' \ x)) \ P \\ &\quad tr_1 = \text{bisimTtrr} \ PP' \ x \end{aligned}$$

The next step towards our goal is to prove that the process R and process R' are bisimilar. We carry out this proof in Agda as follows:

$$\begin{aligned} \text{bisimTraceTrP}_{\infty 3} &: \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P \ P' : \text{Process} \infty \infty \ c) \\ &\quad (PP' : \text{Bisimw} \infty \ \{\infty\} \ P \ P') (l : \text{List} \ (\text{Label} \ lu)) \\ &\quad (\text{tick} : \text{Process} \infty \ \{lu\} \ c \uplus \text{ChoiceSet} \ c) \\ &\quad (tr : \text{TrP} \infty \ l \ \text{tick} \ P') \\ &\rightarrow \text{BisimForNextP} (\text{bisimTraceTrP}_{\infty 1} \ P \ P' \ PP' \ l \ \text{tick} \ tr) \ \text{tick} \\ \text{bisimTraceTrP}_{\infty 3} \ P \ P' \ PP' \ l \ x \ tr &= \text{bisimTraceTrP}_3 (\text{forcep} \ P) \\ &\quad (\text{forcep} \ P') \\ &\quad (\text{forceB} \ PP') \\ &\quad l \ x \ tr \end{aligned}$$

$$\text{bisimTraceTrP}_3 : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$$

```

(P P' : Process ∞ {lu} c)
(PP' : Bisimw {∞} P P')
(l : List (Label lu))
(tick : Process ∞ {lu} c ⊔ ChoiceSet c)
(tr : TrP l tick P')
→ BisimForNextP
(bisimTraceTrP1 P P' PP' l tick tr) tick
bisimTraceTrP3 .(terminate x) (terminate x)
(eqterminate {x} termeqterm)
.[] .(inj2 x) (ter .x) = refl
bisimTraceTrP3 P (terminate x)
(eqterminater termequivP) .[] .(inj2 x)
(ter .x) = termequivPToTick3 x P termequivP
bisimTraceTrP3 .(terminate x) (terminate x)
(eqterminate {x} termeqterm) .[]
.(inj1 (terminate x))
(empty .x) = eqterminate termeqterm
bisimTraceTrP3 P (terminate x)
(eqterminater termequivP) .[]
.(inj1 (terminate x)) (empty .x) =
termequivPToTick3' x P termequivP
bisimTraceTrP3 {lu}{c} .(terminate a) (node P')
(eqterminate {a} (termeqnode terequivP))
l (inj1 Q) (tnode tr) = eqterminate
(termEquivPreservedByTrace+
c l P' Q a terequivP tr)
bisimTraceTrP3 {lu}{c} .(terminate a) (node P')
(eqterminate {a} (termeqnode terequivP))
l (inj2 y) (tnode tr) =
termEquivPr+ c l y P' a terequivP tr
bisimTraceTrP3 .(node P) (node P')
(eqnode {P} PP') l tick
(tnode tr) =
bisimTraceTrP3+ P P' PP' l tick tr

bisimTraceTrP3+ : {lu : LUniv}{c : Choice}
(P P' : Process+ ∞ c)
(PP' : Bisimw+ {∞} P P')
(l : List (Label lu))
(tick : Process ∞ {lu} c ⊔ ChoiceSet c)

```

$$\begin{aligned}
& (tr : \text{TrP} + l \text{ tick } P') \\
& \rightarrow \text{BisimForNextP} \\
& (\text{bisimTraceTrP}_1 + P P' PP' l \text{ tick } tr) \text{ tick} \\
\text{bisimTraceTrP}_3 + & P P' PP' . [] . (\text{inj}_1 (\text{node } P')) \\
& \text{empty} = \text{eqnode } PP' \\
\text{bisimTraceTrP}_3 + & P P' PP' \\
& . (\text{Lab } P' x :: l) (\text{inj}_1 x_1) \\
& (\text{extc } l . (\text{inj}_1 x_1) x x_2) = \\
& \text{bisimTraceTrP}_3 (\text{forcep} \\
& (\text{bisimETraceTrP}_1 + \text{auxmodule.R } P' l x \\
& (\text{inj}_1 x_1) P PP' x_2)) (\text{forcep } (\text{PE } P' x)) \\
& (\text{forceB } (\text{bisimETraceTrP}_1 + \text{auxmodule.RR}' \\
& P' l x (\text{inj}_1 x_1) P PP' x_2)) l \\
& (\text{inj}_1 x_1) x_2 \\
\text{bisimTraceTrP}_3 + & \{lu\}\{c\} P P' PP' . (\text{Lab } P' x :: l) (\text{inj}_2 y) \\
& (\text{extc } l . (\text{inj}_2 y) x x_1) = \\
& \text{bisimTraceTrP}_3 \\
& (\text{forcep } (\text{bisimETraceTrP}_1 + \text{auxmodule.R } P' l x \\
& (\text{inj}_2 y) P PP' x_1)) (\text{forcep } (\text{PE } P' x)) \\
& (\text{forceB } (\text{bisimETraceTrP}_1 + \text{auxmodule.RR}' \\
& P' l x (\text{inj}_2 y) P PP' x_1)) l (\text{inj}_2 y) x_1 \\
\text{bisimTraceTrP}_3 + & \{lu\}\{c\} P P' PP' l (\text{inj}_1 x) \\
& (\text{intc } . l . (\text{inj}_1 x) x_1 x_2) = \\
& \text{bisimTraceTrP}_3 \\
& (\text{forcep } (\text{bisimITraceTrP}_1 + \text{auxmodule.R } l \\
& (\text{inj}_1 x) P' P PP' x_1 x_2)) \\
& (\text{forcep } (\text{PI } P' x_1)) \\
& (\text{forceB } (\text{bisimITraceTrP}_1 + \text{auxmodule.RR}' l \\
& (\text{inj}_1 x) P' P PP' x_1 x_2)) l (\text{inj}_1 x) x_2 \\
\text{bisimTraceTrP}_3 + & \{lu\}\{c\} P P' PP' l (\text{inj}_2 y) (\text{intc } . l . (\text{inj}_2 y) x x_1) = \\
& \text{bisimTraceTrP}_3 \\
& (\text{forcep } (\text{bisimITraceTrP}_1 + \text{auxmodule.R } l (\text{inj}_2 y) P' \\
& P PP' x x_1)) \\
& (\text{forcep } (\text{PI } P' x)) \\
& (\text{forceB } (\text{bisimITraceTrP}_1 + \text{auxmodule.RR}' l (\text{inj}_2 y) \\
& P' P PP' x x_1)) l (\text{inj}_2 y) x_1 \\
\text{bisimTraceTrP}_3 + & \{lu\}\{c\} P P' PP' . [] . (\text{inj}_2 (\text{PT } P' x)) (\text{terc } x) = \text{refl}
\end{aligned}$$

By $R \sim R' \wedge \text{stable}(R')$ we get R is non-divergent process. The proof in Agda as follows:

mutual

$$\begin{aligned}
& \text{bisimStableImpliesNotDivergent}\infty : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P \ P' : \text{Process}\infty \ \infty \ \{lu\} \ c) \\
& \quad (PP' : \text{Bisimw}\infty \ P \ P') \\
& \quad (PS' : \text{stable}\infty \ P') \\
& \quad (nonDivP' : \text{NonDivergent}\infty \ P') \\
& \rightarrow \text{NonDivergent}\infty \ P \\
& \text{forceND} (\text{bisimStableImpliesNotDivergent}\infty \ c \ P \ P' \ PP' \ PS' \ nonDivP') \\
& \quad = \text{bisimStableImpliesNotDivergent} \ c \ (\text{forcep} \ P) \\
& \quad \quad (\text{forcep} \ P') \\
& \quad \quad (\text{forceB} \ PP') \\
& \quad \quad PS' \ (\text{forceND} \ nonDivP') \\
\\
& \text{bisimStableImpliesNotDivergent} : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\
& \quad (PP' : \text{Bisimw} \ P \ P') \\
& \quad (PS' : \text{stable} \ P') \\
& \quad (nonDivP' : \text{NonDivergent} \ P') \\
& \rightarrow \text{NonDivergent} \ P \\
& \text{bisimStableImpliesNotDivergent} \ c \ (\text{terminate} \ x) \ P' \ PP' \ PS' \ nonDivP' = \text{tt} \\
& \text{bisimStableImpliesNotDivergent} \ c \ (\text{node} \ P) \ (\text{terminate} \ a) \\
& \quad (\text{eqterminater} \ (\text{termeqnode} \ terequivP)) \\
& \quad PS' \ nonDivP' = \\
& \quad \text{TerImpliesNotDivergentaux} \ c \ (\text{node} \ P) \\
& \quad \quad a \ ((\text{termeqnode} \ terequivP)) \\
\\
& \text{bisimStableImpliesNotDivergent} \ c \ (\text{node} \ P) \ (\text{node} \ P') \ (\text{eqnode} \ PP') \ PS' \\
& \quad nonDivP' = \text{bisimStableImpliesNotDivergent}+ \\
& \quad \quad c \ P \ P' \ PP' \ PS' \ nonDivP' \\
\\
& \text{bisimStableImpliesNotDivergent}+ : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P \ P' : \text{Process}+ \ \infty \ \{lu\} \ c) \\
& \quad (PP' : \text{Bisimw}+ \ P \ P') \\
& \quad (PS' : \text{stable}+ \ P') \\
& \quad (nonDivP' : \text{NonDivergent}+ \ P') \\
& \rightarrow \text{NonDivergent}+ \ P \\
& \text{bisimStableImpliesNotDivergent}+ \ c \ P \ P' \ PP' \ PS' \ nonDivP' \\
& \quad = \text{nondiv}+r \ PP' \ nonDivP'
\end{aligned}$$

Therefore there exists \hat{R} such that $R \Rightarrow \hat{R} \wedge \text{stable}(\hat{R})$. In Agda we split

this in three parts: we define \hat{R} , then a trace from R to \hat{R} , and finally a proof that \hat{R} is Schneider stable. That we obtain Schneider stability will be enough: When using this lemma in order to prove the Main Lemma 2.4.6 later, we will get that the Schneider stable process we obtain is DRW-bisimilar to a process which is Roscoe stable. We will then show that if a Schneider stable process is DRW-bisimilar to a Roscoe stable process, then it is actually Roscoe stable. So the process in question will in fact already be as well Roscoe stable.

The proof of the above statement in Agda is as follows:

mutual

```

nonDivBecomeStable $\infty_1$  : {lu : LUniv}{c : Choice}
  (P : Process $\infty$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $\infty$  P)
  → Process  $\infty$  {lu} c
nonDivBecomeStable $\infty_1$  c P nonDivP = nonDivBecomeStable $_1$ 
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable $\infty_2$  : {lu : LUniv}{c : Choice}
  (P : Process $\infty$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $\infty$  P)
  →
    TrP $\infty$  {lu} [] (inj $_1$ 
      (nonDivBecomeStable $\infty_1$  c P nonDivP)) P
nonDivBecomeStable $\infty_2$  c P nonDivP = nonDivBecomeStable $_2$ 
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable $\infty_3$  : {lu : LUniv}{c : Choice}
  (P : Process $\infty$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $\infty$  P)
  → stableSch (nonDivBecomeStable $\infty_1$ 
    c P nonDivP)
nonDivBecomeStable $\infty_3$  c P nonDivP = nonDivBecomeStable $_3$ 
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable $+$  $_1$  : {lu : LUniv}{c : Choice}
  (P : Process $+$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $+$  P)
  →
    Process  $\infty$  {lu} c
nonDivBecomeStable $+$  $_1$  c P (nondiv x (inj $_1$  int)) =
  nonDivBecomeStable $\infty_1$ 
    c (PI P int) (x int)
nonDivBecomeStable $+$  $_1$  c P (nondiv x (inj $_2$  stab)) = node P

```

$$\begin{aligned}
& \text{nonDivBecomeStable}+2 : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P : \text{Process}+ \infty \{lu\} c) \\
& \quad (\text{nonDiv}P : \text{NonDivergent}+ P) \\
& \rightarrow \text{TrP}+ \{lu\} [] (\text{inj}_1 \\
& \quad (\text{nonDivBecomeStable}+1 c P \text{nonDiv}P)) P \\
& \text{nonDivBecomeStable}+2 c P (\text{nondiv } x (\text{inj}_1 \text{int})) = \text{intc } [] (\text{inj}_1 \\
& \quad (\text{nonDivBecomeStable}+1 c P \\
& \quad (\text{nondiv } x (\text{inj}_1 \text{int})))) \text{int} \\
& \quad (\text{nonDivBecomeStable}\infty_2 c \\
& \quad (\text{PI } P \text{int}) (x \text{int})) \\
& \text{nonDivBecomeStable}+2 c P (\text{nondiv } x (\text{inj}_2 \text{stab})) = \text{empty}
\end{aligned}$$

$$\begin{aligned}
& \text{nonDivBecomeStable}+3 : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P : \text{Process}+ \infty \{lu\} c) \\
& \quad (\text{nonDiv}P : \text{NonDivergent}+ P) \\
& \rightarrow \text{stableSch} \\
& \quad (\text{nonDivBecomeStable}+1 c P \text{nonDiv}P) \\
& \text{nonDivBecomeStable}+3 c P (\text{nondiv } x (\text{inj}_1 \text{int})) = \\
& \quad \text{nonDivBecomeStable}\infty_3 c \\
& \quad (\text{PI } P \text{int}) (x \text{int}) \\
& \text{nonDivBecomeStable}+3 c P (\text{nondiv } x (\text{inj}_2 \text{stab})) = \text{stab}
\end{aligned}$$

$$\begin{aligned}
& \text{nonDivBecomeStable}_1 : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P : \text{Process} \infty \{lu\} c) \\
& \quad (\text{nonDiv}P : \text{NonDivergent } P) \\
& \rightarrow \text{Process} \infty \{lu\} c \\
& \text{nonDivBecomeStable}_1 c (\text{terminate } x) \text{nonDiv}P = \text{terminate } x \\
& \text{nonDivBecomeStable}_1 c (\text{node } x) \text{nonDiv}P = \\
& \quad \text{nonDivBecomeStable}+1 c x \text{nonDiv}P
\end{aligned}$$

$$\begin{aligned}
& \text{nonDivBecomeStable}_2 : \{lu : \text{LUniv}\}(c : \text{Choice}) \\
& \quad (P : \text{Process} \infty \{lu\} c) \\
& \quad (\text{nonDiv}P : \text{NonDivergent } P) \\
& \rightarrow \text{TrP } \{lu\} [] (\text{inj}_1 \\
& \quad (\text{nonDivBecomeStable}_1 c P \text{nonDiv}P)) P \\
& \text{nonDivBecomeStable}_2 c (\text{terminate } x) \text{nonDiv}P = \text{empty } x \\
& \text{nonDivBecomeStable}_2 c (\text{node } x) \text{nonDiv}P \\
& \quad = \text{tnode } (\text{nonDivBecomeStable}+2 c x \text{nonDiv}P)
\end{aligned}$$

$\text{nonDivBecomeStable}_3 : \{lu : \text{LUniv}\} \{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c)$
 $(\text{nonDiv}P : \text{NonDivergent } P)$
 $\rightarrow \text{stableSch } (\text{nonDivBecomeStable}_1 c P \text{ nonDiv}P)$
 $\text{nonDivBecomeStable}_3 c (\text{terminate } x) \text{ nonDiv}P = _$
 $\text{nonDivBecomeStable}_3 c (\text{node } x) \text{ nonDiv}P = \text{nonDivBecomeStable}+_3 c$
 $x \text{ nonDiv}P$

By $R \sim R'$ there exists \hat{R}' such that $R' \Rightarrow \hat{R}'$ such that $\hat{R} \sim \hat{R}'$.
 By $\text{stable}(R')$ we get $R' = \hat{R}'$, therefore we get $\hat{R} \sim R'$.
 In the following we carry out the proof in Agda:

mutual

$\text{bisimPPWithEmptyTr}_\infty : \{lu : \text{LUniv}\} \{c : \text{Choice}\}$
 $(P P' : \text{Process}_\infty \infty \{lu\} c)$
 $(PP' : \text{Bisimw}_\infty P P') (PS' : \text{stable}_\infty P')$
 $(\text{nonDiv}P : \text{NonDivergent}_\infty P)$
 $(tr : \text{TrP}_\infty \{lu\} [])$
 $(\text{inj}_1 (\text{nonDivBecomeStable}_\infty_1 c P \text{ nonDiv}P)) P)$
 $\rightarrow \text{Bisimw } (\text{nonDivBecomeStable}_\infty_1 c P \text{ nonDiv}P)$
 $(\text{forceP } P')$
 $\text{bisimPPWithEmptyTr}_\infty P P' PP' PS' \text{ nonDiv}P tr =$
 $\text{bisimPPWithEmptyTr } (\text{forceP } P) (\text{forceP } P')$
 $(\text{forceB } PP') PS' (\text{forceND } \text{nonDiv}P) tr$

$\text{bisimPPWithEmptyTr} : \{lu : \text{LUniv}\} \{c : \text{Choice}\}$
 $(P P' : \text{Process} \infty \{lu\} c)$
 $(PP' : \text{Bisimw } P P') (PS' : \text{stable } P')$
 $(\text{nonDiv}P : \text{NonDivergent } P)$
 $(tr : \text{TrP } \{lu\} []) (\text{inj}_1$
 $(\text{nonDivBecomeStable}_1 \{lu\} c P \text{ nonDiv}P)) P)$
 $\rightarrow \text{Bisimw } (\text{nonDivBecomeStable}_1 \{lu\} c P \text{ nonDiv}P) P'$
 $\text{bisimPPWithEmptyTr } \{lu\} \{c\} . (\text{terminate } x) (\text{terminate } x_1)$
 $PP' PS' \text{ nonDiv}P (\text{empty } x) = PP'$
 $\text{bisimPPWithEmptyTr } (\text{node } P) (\text{terminate } a)$
 $(\text{eqterminater } (\text{termeqnode } \text{terequiv}P))$
 $PS' (\text{nondiv } \text{nondivPI } (\text{inj}_1 x)) (\text{tnode } tr) =$
 $\text{nonDivBecomesStableBisimProof}_\infty (\text{PI } P x)$

$$\begin{aligned}
& (\text{nondivPI } x) \text{ a } (\text{onlyIntChoice } \text{terequivP } x) \\
\text{bisimPPWithEmptyTr } & (\text{node } P) (\text{terminate } x) \\
& (\text{eqterminater } (\text{termeqnode } \text{terequivP})) \\
& PS' (\text{nondiv } x_1 (\text{inj}_2 y)) (\text{tnode } tr) = \\
& \text{eqterminater } (\text{termeqnode } \text{terequivP}) \\
\text{bisimPPWithEmptyTr } & (\text{terminate } P) (\text{node } P') PP' PS' \text{nonDivP } tr = \\
& PP' \\
\text{bisimPPWithEmptyTr } & (\text{node } P) (\text{node } P') (\text{eqnode } \text{bisimPP}') PS' \\
& (\text{nondiv } x \text{chemptyornot}) (\text{tnode } tr) = \\
& \text{bisimPPWithEmptyTr+ } P P' \text{bisimPP}' \\
& PS' (\text{nondiv } x \text{chemptyornot}) tr
\end{aligned}$$

$$\begin{aligned}
\text{bisimPPWithEmptyTr+ } & : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\
& (P P' : \text{Process+ } \infty \{lu\} c) \\
& (PP' : \text{Bisimw+ } P P') (PS' : \text{stable+ } P') \\
& (\text{nonDivP} : \text{NonDivergent+ } P) \\
& (tr : \text{TrP+ } \{lu\} [] (\text{inj}_1 \\
& (\text{nonDivBecomeStable+}_1 c P \text{nonDivP}))) P) \\
& \rightarrow \text{Bisimw } (\text{nonDivBecomeStable+}_1 c P \text{nonDivP}) \\
& (\text{node } P') \\
\text{bisimPPWithEmptyTr+ } & \{lu\} \{c\} P P' PP' PS' \\
& (\text{nondiv } \text{nondiv}' (\text{inj}_1 x_1)) tr = PP''
\end{aligned}$$

where

$$\begin{aligned}
P' \sim & : \text{Process}\infty \infty \{lu\} c \\
P' \sim & = \text{bisimIP}' PP' x_1
\end{aligned}$$

$$\begin{aligned}
\text{trP}'P' \sim & : P' \rightarrow +^* [[]] (\text{forcep } (P' \sim)) \\
\text{trP}'P' \sim & = \text{bisimltr } PP' x_1
\end{aligned}$$

$$\begin{aligned}
P' \equiv P' \sim & : \text{node } P' \equiv \text{forcep } P' \sim \{\infty\} \\
P' \equiv P' \sim & = \text{emptyTrPtoQImpliesEq+ } P' \\
& (\text{forcep } P' \sim) PS' \text{trP}'P' \sim
\end{aligned}$$

$$\begin{aligned}
P' \sim \equiv P' & : \text{forcep } P' \sim \{\infty\} \equiv \text{node } P' \\
P' \sim \equiv P' & \text{rewrite } P' \equiv P' \sim = \text{refl}
\end{aligned}$$

$$\begin{aligned}
P' \sim \text{stable} & : \text{stable } (\text{forcep } P' \sim) \\
P' \sim \text{stable} & \text{rewrite } P' \sim \equiv P' = PS'
\end{aligned}$$



$$\begin{aligned}
PP'' &: \text{Bisimw} \left(\text{nonDivBecomeStable}_1 \ c \right. \\
&\quad \left(\text{forcep} \ (\text{PI} \ P \ x_1) \right) \left(\text{forceND} \ (\text{nondiv}' \ x_1) \right) \left. \right) \\
&\quad \left(\text{forcep} \ P' \sim \right) \\
PP'' &= \text{bisimPPWithEmptyTr} \left(\text{forcep} \ (\text{PI} \ P \ x_1) \right) \\
&\quad \left(\text{forcep} \ P' \sim \ \{\infty\} \right) \\
&\quad \left(\text{forceB} \ (\text{bisimInext} \ PP' \ x_1) \right) \\
&\quad \quad P' \sim \text{stable} \ (\text{forceND} \ (\text{nondiv}' \ x_1)) \\
&\quad \left(\text{nonDivBecomeStable}_2 \ c \right. \\
&\quad \left. \left(\text{forcep} \ (\text{PI} \ P \ x_1) \right) \left(\text{forceND} \ (\text{nondiv}' \ x_1) \right) \right) \\
\\
PP''' &: \text{Bisimw} \left(\text{nonDivBecomeStable}_{\infty_1} \ c \right. \\
&\quad \left(\text{PI} \ P \ x_1 \right) \left(\text{nondiv}' \ x_1 \right) \left(\text{node} \ P' \right) \\
PP''' &\text{ rewrite } P' \equiv P' \sim = PP'' \\
\\
&\text{bisimPPWithEmptyTr} + P \ P' \ PP' \ PS' \\
&\quad \left(\text{nondiv} \ x \ (\text{inj}_2 \ y) \right) \text{empty} = \text{eqnode} \ PP' \\
&\text{bisimPPWithEmptyTr} + P \ P' \ PP' \ PS' \\
&\quad \left(\text{nondiv} \ x \ (\text{inj}_2 \ y) \right) \\
&\quad \quad \left(\text{intc} \ .[] \ .(\text{inj}_1 \right. \\
&\quad \quad \quad \left. \left(\text{node} \ P \right) \right) x_1 \ x_2) = \text{eqnode} \ PP'
\end{aligned}$$

10.9 Proof of Lemma 2.4.6, Part 2 (Obtaining Roscoe Stability)

In the previous Section 10.8 we proved that the resulting process was Schneider stable rather than Roscoe stable. This is sufficient for the proofs in the following. However, we will show that in fact Roscoe stability is obtained. The reason is that the process we obtained was Schneider stable and DRW-bisimilar to a Roscoe stable process. These two conditions imply that the process is actually Roscoe stable.

First we show that if two processes are DRW-bisimilar, and one is Roscoe-stable and the other one is Schneider stable, then the Schneider one is actually Roscoe stable:

$$\text{stabSchBisim2stabRosclsStabRosc} : \{lu : \text{LUniv}\} \{c : \text{Choice}\}$$



```

(P P' : Process ∞ {lu} c)
(PP' : Bisimw P P')
(stabP' : stable P')
(stabSchP : stableSch P)
→ stable P

stabSchBisim2stabRosclsStabRosc
  (terminate x) (terminate x₁)
  (eqterminate terequiv) stabP' stabSchP = stabP'
stabSchBisim2stabRosclsStabRosc
  (terminate x) (node P')
  (eqterminate (termegnode terequivP'))
  (P' stabSch „ notickP') stabSchP _
  = tauOrTickNoTauP'ImpliesConclusion tauOrTickNoTauP' where
    tauOrTickNoTauP' : ChoiceSet (I P') ⊔ (¬ (ChoiceSet (I P'))
      × ChoiceSet (T P'))
    tauOrTickNoTauP' = hasTauOrTickNoTau terequivP'

    tauOrTickNoTauP'ImpliesConclusion : ChoiceSet (I P')
      ⊔ (¬ (ChoiceSet (I P'))
        × ChoiceSet (T P')) → ⊥

    tauOrTickNoTauP'ImpliesConclusion (inj₁ tauChoiceP')
      = P' stabSch tauChoiceP'
    tauOrTickNoTauP'ImpliesConclusion (inj₂ (– „ tickChoiceP'))
      = notickP' tickChoiceP'

stabSchBisim2stabRosclsStabRosc
  (terminate x) . (terminate –) (eqterminater terequiv)
  stabP' stabSchP
  = stabP'
stabSchBisim2stabRosclsStabRosc
  (node P) (terminate x) PP' stabP' stabSchP
  = ⊥-elim (stabP' –)
stabSchBisim2stabRosclsStabRosc
  (node P) (node P') (eqnode bisimQQ') (stabSchP' „ noTickP')
  stabSchP
  = stabSchP „ noTickP
  where
    traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj₂ (PT P t)) P'
    traceToTickP = bisimTtr bisimQQ'

```

```

noTickP : ¬ (ChoiceSet (T P))
noTickP t = schStabNoTraceToInj2+ P' stabSchP' (PT P t)
          (traceToTickP t) noTickP'

```

Now we show Lemma 2.4.6 in full: If we have two processes P , P' , which are DRW-bisimilar, and P' has a trace to Q' which is stable, then we find a process called $\hat{Q} := \text{bisimTraceTrP}\infty\text{Qhat}$, a trace $\text{bisimTraceTrP}\infty\text{trhat}_2$ from P to \hat{Q} , such that \hat{Q} and Q' are bisimilar (proof $\text{bisimTraceTrP}\infty\text{QhatQ}'$), and \hat{Q} is stable (proof $\text{bisimTraceTrP}\infty\text{stabQhat}$):

Theorem 10.9.1 (Agda Theorem corresponding to Lemma 2.4.6)

```

module _ {lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ c)
  (PP' : Bisimw∞ {∞} P P')(l : List (Label lu))
  (Q' : Process ∞ {lu} c)
  (tr' : TrP∞ l (inj1 Q') P')
  (stab' : stable Q') where

  bisimTraceTrP∞Qhat : Process ∞ {lu} c

  bisimTraceTrP∞QhatQ' : Bisimw bisimTraceTrP∞Qhat Q'

  bisimTraceTrP∞stabQhat : stable bisimTraceTrP∞Qhat

  bisimTraceTrP∞trhat2 : TrP∞ {lu} l (inj1 bisimTraceTrP∞Qhat) P

```

Proof:

```

module _ {lu : LUniv}{c : Choice}(P P' : Process∞ ∞ c)
  (PP' : Bisimw∞ {∞} P P')(l : List (Label lu))
  (Q' : Process ∞ {lu} c)
  (tr' : TrP∞ l (inj1 Q') P')
  (stab' : stable Q') where

  bisimTraceTrP∞Qcom : Process ∞ c ⊔ ChoiceSet c
  bisimTraceTrP∞Qcom = bisimTraceTrP∞1 P P' PP' l (inj1 Q') tr'

  bisimTraceTrP∞trcom : TrP∞ {lu} l (bisimTraceTrP∞1 P P' PP' l

```

$$\text{bisimTraceTrP}_{\infty \text{trcom}} = \text{bisimTraceTrP}_{\infty 2} P P' PP' l (\text{inj}_1 Q') tr'$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty QQ' \text{com}} &: \text{BisimForNextP} (\text{bisimTraceTrP}_{\infty 1} P P' PP' l \\ &\quad (\text{inj}_1 Q') tr') (\text{inj}_1 Q') \\ \text{bisimTraceTrP}_{\infty QQ' \text{com}} &= \text{bisimTraceTrP}_{\infty 3} P P' PP' l (\text{inj}_1 Q') \quad tr' \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty Q} &: \text{Process} \infty \{lu\} c \\ \text{bisimTraceTrP}_{\infty Q} &= \text{lemmayyy}_1' \text{bisimTraceTrP}_{\infty Q \text{com}} Q' \text{stab}' \\ &\quad \text{bisimTraceTrP}_{\infty QQ' \text{com}} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty \text{tr}} &: \text{TrP}_{\infty} \{lu\} l (\text{inj}_1 \text{bisimTraceTrP}_{\infty Q}) P \\ \text{bisimTraceTrP}_{\infty \text{tr}} &= \text{lemmayyy}_2' (\text{bisimTraceTrP}_{\infty Q \text{com}}) l (\text{forcep } P) \\ &\quad Q' \text{stab}' \text{bisimTraceTrP}_{\infty QQ' \text{com}} \\ &\quad \text{bisimTraceTrP}_{\infty \text{trcom}} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty QQ'} &: \text{Bisimw} \text{bisimTraceTrP}_{\infty Q} Q' \\ \text{bisimTraceTrP}_{\infty QQ'} &= \text{lemmayyy}_3' \\ &\quad \text{bisimTraceTrP}_{\infty Q \text{com}} Q' \text{stab}' \\ &\quad \text{bisimTraceTrP}_{\infty QQ' \text{com}} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty Q \text{hat}} &: \text{Process} \infty \{lu\} c \\ \text{bisimTraceTrP}_{\infty Q \text{hat}} &= \text{nonDivBecomeStable}_1 c \text{bisimTraceTrP}_{\infty Q} \\ &\quad (\text{bisimStableImpliesNotDivergent } c \\ &\quad \text{bisimTraceTrP}_{\infty Q} Q' \\ &\quad \text{bisimTraceTrP}_{\infty QQ'} \text{stab}' \\ &\quad (\text{stableImpliesNonDiv } Q' \text{stab}')) \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty \text{trhat}} &: \text{TrP} \{lu\} [] (\text{inj}_1 \text{bisimTraceTrP}_{\infty Q \text{hat}}) \\ &\quad \text{bisimTraceTrP}_{\infty Q} \\ \text{bisimTraceTrP}_{\infty \text{trhat}} &= \text{nonDivBecomeStable}_2 c \\ &\quad \text{bisimTraceTrP}_{\infty Q} \\ &\quad (\text{bisimStableImpliesNotDivergent } c \end{aligned}$$

$$\begin{aligned} & \text{bisimTraceTrP}_{\infty} Q' \\ & \text{bisimTraceTrP}_{\infty} Q Q' \text{ stab}' \\ & (\text{stableImpliesNonDiv } Q' \text{ stab}') \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty} \text{Qhat} Q' & : \text{Bisimw } \text{bisimTraceTrP}_{\infty} \text{Qhat } Q' \\ \text{bisimTraceTrP}_{\infty} \text{Qhat} Q' & = \text{bisimPPWithEmptyTr } \text{bisimTraceTrP}_{\infty} Q Q' \\ & \quad \text{bisimTraceTrP}_{\infty} Q Q' \text{ stab}' \\ & \quad (\text{bisimStableImpliesNotDivergent } c \\ & \quad \quad \text{bisimTraceTrP}_{\infty} Q Q' \\ & \quad \quad \text{bisimTraceTrP}_{\infty} Q Q' \text{ stab}' \\ & \quad \quad (\text{stableImpliesNonDiv } Q' \text{ stab}')) \\ & \text{bisimTraceTrP}_{\infty} \text{trhat} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty} \text{stabSchQhat} & : \text{stableSch } \text{bisimTraceTrP}_{\infty} \text{Qhat} \\ \text{bisimTraceTrP}_{\infty} \text{stabSchQhat} & = \text{nonDivBecomeStable}_3 \ c \ \text{bisimTraceTrP}_{\infty} Q \\ & \quad (\text{bisimStableImpliesNotDivergent } c \\ & \quad \quad \text{bisimTraceTrP}_{\infty} Q Q' \\ & \quad \quad \text{bisimTraceTrP}_{\infty} Q Q' \text{ stab}' \\ & \quad \quad (\text{stableImpliesNonDiv } Q' \text{ stab}')) \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty} \text{stabQhat} & : \text{stable } \text{bisimTraceTrP}_{\infty} \text{Qhat} \\ \text{bisimTraceTrP}_{\infty} \text{stabQhat} & = \text{stabSchBisim2stabRosclsStabRosc} \\ & \quad \text{bisimTraceTrP}_{\infty} \text{Qhat} \\ & \quad Q' \text{ bisimTraceTrP}_{\infty} \text{Qhat} Q' \text{ stab}' \\ & \quad \text{bisimTraceTrP}_{\infty} \text{stabSchQhat} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty} \text{trhat}_1 & : \text{TrP}_{\infty} \{lu\} (l ++ []) \\ & \quad (\text{inj}_1 \text{ bisimTraceTrP}_{\infty} \text{Qhat}) P \\ \text{bisimTraceTrP}_{\infty} \text{trhat}_1 & = \text{trPAppendTrw}_{\infty} \ c \ P \ \text{bisimTraceTrP}_{\infty} Q \ l \ [] \\ & \quad (\text{inj}_1 \text{ bisimTraceTrP}_{\infty} \text{Qhat}) \\ & \quad \text{bisimTraceTrP}_{\infty} \text{tr} \ \text{bisimTraceTrP}_{\infty} \text{trhat} \end{aligned}$$

$$\begin{aligned} \text{bisimTraceTrP}_{\infty} \text{trhat}_2 & : \text{TrP}_{\infty} \{lu\} l (\text{inj}_1 \text{ bisimTraceTrP}_{\infty} \text{Qhat}) P \\ \text{bisimTraceTrP}_{\infty} \text{trhat}_2 & = \text{subst } (\lambda l' \rightarrow \text{TrP}_{\infty} \{lu\} l') \end{aligned}$$

```

                                (inj1 bisimTraceTrP∞Qhat)
                                P)
                                eql bisimTraceTrP∞trhat1 where

eql : (l ++ [] ) ≡ l
eql = lemEqList l

```

We show that if we drop the condition that Q' is stable, then we find a process Q which is reachable by the same trace and DRW-bisimilar to Q :

Theorem 10.9.2 (Agda Theorem)

```

module _ {lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ c)
  (PP' : Bisimw∞ {∞} P P')
  (l : List (Label lu))
  (Q' : Process ∞ {lu} c)
  (tr' : TrP∞ l (inj1 Q') P') where

bisimTraceTrP∞Qcom : Process ∞ c ⊔ ChoiceSet c
bisimTraceTrP∞trcom : TrP∞ {lu} l (bisimTraceTrP∞1 P P' PP' l
                                (inj1 Q') tr') P
bisimTraceTrP∞QQ'com : BisimForNextP (bisimTraceTrP∞1 P P' PP' l
                                (inj1 Q') tr') (inj1 Q')

```

Proof:

```

bisimTraceTrP∞Qcom = bisimTraceTrP∞1 P P' PP' l (inj1 Q') tr'
bisimTraceTrP∞trcom = bisimTraceTrP∞2 P P' PP' l (inj1 Q') tr'
bisimTraceTrP∞QQ'com = bisimTraceTrP∞3 P P' PP' l (inj1 Q') tr'

```

10.10 Bisimilarity Implies Stable Failures Equivalence

One of the main tools for proving equivalence with respect to stable failures semantics is to a proof that strong and DRW bisimilarity implies stable failures equivalence. The reason is that it is usually much easier to prove strong or DRW bisimilarity than to prove directly stable failures equivalence. Proving stable failures equivalence is difficult since one needs to derive corresponding lemma for any future processes, which might be of different shape than

the original law. When proving DRW-bisimilarity, these auxiliary lemma occur much more naturally, and are much easier to deal with.

In this section we show that DRW and strong bisimilarity imply stable failures equivalence. We first show that DRW bisimilarity implies stable failures refinement. Using symmetry of bisimilarity we obtain as well stable failures equivalence, and using the fact that strong bisimilarity implies DRW bisimilarity, we obtain those laws for strong bisimilarity as well.

In order to prove that weak bisimilarity implies stable failure refinement we first prove refinement with respect to \sqsubseteq_{sf_1} . For this we assume we have a trace leading to a stable failure for P' and get a proof for P . We use here **where** clauses to build up the proof in stages by reflecting each step for P' by a corresponding step for P and maintaining bisimilarity between the corresponding processes. The full proof is as follows:

$$\begin{aligned}
 &\text{bisimwImplies}\sqsubseteq_{sf_1} : \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\
 &\quad (P' : \text{Process} \infty \{lu\} c) \\
 &\quad (PP' : \text{Bisimw} \{\infty\} P P') \\
 &\quad \rightarrow P \sqsubseteq_{sf_1} P' \\
 &\text{bisimwImplies}\sqsubseteq_{sf_1} \{lu\}\{c\} P P' PP' l X \\
 &\quad (\text{stableFp } Q' \text{ tr}' \text{ stab}' \text{ drefuse}') \\
 &\quad = (\text{stableFp } Qhat \text{ trhat}_2 \\
 &\quad \quad (\quad \text{stabSchNoTickIfRos2StablePar } Qhat \\
 &\quad \quad \quad \text{true stabSchQhat stabNoTick} \quad) \\
 &\quad \quad \text{drefusehat}) \\
 &\text{where} \\
 &\quad Qcom : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c \\
 &\quad Qcom = \text{bisimTraceTrP}_1 P P' PP' l (\text{inj}_1 Q') \text{ tr}' \\
 &\quad \text{trcom} : \text{TrP } \{lu\} l (\text{bisimTraceTrP}_1 P P' PP' l \\
 &\quad \quad (\text{inj}_1 Q') \text{ tr}') P \\
 &\quad \text{trcom} = \text{bisimTraceTrP}_2 P P' PP' l (\text{inj}_1 Q') \text{ tr}' \\
 &\quad QQ'com : \text{BisimForNextP} (\text{bisimTraceTrP}_1 P P' PP' l \\
 &\quad \quad (\text{inj}_1 Q') \text{ tr}') (\text{inj}_1 Q') \\
 &\quad QQ'com = \text{bisimTraceTrP}_3 P P' PP' l (\text{inj}_1 Q') \quad \text{tr}' \\
 &\quad Q : \text{Process} \infty \{lu\} c \\
 &\quad Q = \text{lemmayyy}_1 Qcom Q' \text{ stab}' X \text{ drefuse}' QQ'com \\
 &\quad \text{tr} : \quad \text{TrP } \{lu\} l (\text{inj}_1 Q) P \\
 &\quad \text{tr} = \quad \text{lemmayyy}_2 Qcom l P Q' \text{ stab}' X \text{ drefuse}' QQ'com \text{ trcom}
 \end{aligned}$$

$QQ' : \text{Bisimw } Q \ Q'$
 $QQ' = \text{lemmayy}_3 \ Q \text{com } Q' \text{stab}' \ X \ \text{drefuse}' \ QQ' \text{com}$

$\text{Qhat} : \text{Process} \ \infty \ \{lu\} \ c$
 $\text{Qhat} = \text{nonDivBecomeStable}_1 \ c \ Q$
 $\quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ \text{stab}'$
 $\quad (\text{stableImpliesNonDiv } Q' \ \text{stab}'))$

$\text{trhat} : \text{TrP } \{lu\} \ [] \ (\text{inj}_1 \ \text{Qhat}) \ Q$
 $\text{trhat} = \text{nonDivBecomeStable}_2 \ c \ Q$
 $\quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ \text{stab}'$
 $\quad (\text{stableImpliesNonDiv } Q' \ \text{stab}'))$

$\text{QhatQ}' : \quad \text{Bisimw } \text{Qhat} \ Q'$
 $\text{QhatQ}' = \quad \text{bisimPPWithEmptyTr } Q \ Q' \ QQ' \ \text{stab}'$
 $\quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ \text{stab}'$
 $\quad (\text{stableImpliesNonDiv } Q' \ \text{stab}')) \ \text{trhat}$

$\text{stabSchQhat} : \text{stableSch } \text{Qhat}$
 $\text{stabSchQhat} = \text{nonDivBecomeStable}_3 \ c \ Q$
 $\quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ \text{stab}'$
 $\quad (\text{stableImpliesNonDiv } Q' \ \text{stab}'))$

$\text{stabNoTick} : \text{noTickIfRoscoe } \text{true} \ \text{Qhat}$
 $\text{stabNoTick} = \text{bisimwStableToNoTick } \text{Qhat} \ Q' \ \text{QhatQ}' \ \text{stab}' \ \text{stabSchQhat}$

$\text{trhat}_1 : \text{TrP } \{lu\} \ (l \ ++ \ []) \ (\text{inj}_1 \ \text{Qhat}) \ P$
 $\text{trhat}_1 = \text{trPAppendTrw } c \ P \ Q \ l \ [] \ (\text{inj}_1 \ \text{Qhat}) \ \text{tr } \text{trhat}$

$\text{eqI} : \quad (l \ ++ \ []) \equiv l$
 $\text{eqI} = \text{lemEqList } l$

$\text{trhat}_2 : \text{TrP } \{lu\} \ l \ (\text{inj}_1 \ \text{Qhat}) \ P$
 $\text{trhat}_2 = \text{subst } (\lambda \ l' \rightarrow \text{TrP } \{lu\} \ l' \ (\text{inj}_1 \ \text{Qhat}) \ P) \ \text{eqI} \ \text{trhat}_1$

$\text{drefusehat} : \quad \text{DRefusal } \text{Qhat} \ \text{true} \ X$
 $\text{drefusehat} = \quad \text{bisimDRefusal } Q' \ \text{Qhat} \ \text{stab}'$
 $\quad (\text{BismwSym } \text{Qhat} \ Q' \ \text{QhatQ}') \ X \ \text{true} \ \text{drefuse}'$

There are similar proofs for and Process_∞ .



Now we combine the proofs that DRW bisimilarity implies trace refinement with the proof above to get a proof that DRW bisimilarity implies stable failures refinement:

$$\begin{aligned} \text{bisimwImplies}\sqsubseteq\text{sf}_1+ : & \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c) \\ & (P' : \text{Process}+ \infty \{lu\} c) \\ & (PP' : \text{Bisimw}+ \{\infty\} P P') \\ \rightarrow & P \sqsubseteq\text{sf}_1+ P' \end{aligned}$$

Next we show that we obtain the refinement statement with P and P' interchanged. Here we use the proof that weak bisimilarity is symmetric:

$$\begin{aligned} \text{bisimwImplies}\sqsubseteq\text{sf}_1r+ : & \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c) \\ & (P' : \text{Process}+ \infty \{lu\} c) \\ & (PP' : \text{Bisimw}+ \{\infty\} P P') \\ \rightarrow & P' \sqsubseteq\text{sf}_1+ P \\ \text{bisimwImplies}\sqsubseteq\text{sf}_1r+ P P' PP' = & \\ \text{bisimwImplies}\sqsubseteq\text{sf}_1+ P' P & \\ (\text{BismwSym}+ P P' PP') & \end{aligned}$$

Now we combine the two proofs to a proof of stale failure equivalence:

Theorem 10.10.1 (Agda Theorem)

$$\begin{aligned} \text{bisimwImplies}=\text{sf} : & \{lu : \text{LUniv}\}\{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} c) \\ & (P' : \text{Process} \infty \{lu\} c) \\ & (PP' : \text{Bisimw} \{\infty\} P P') \\ \rightarrow & P =\text{sf} P' \end{aligned}$$

Proof:

$$\begin{aligned} \text{bisimwImplies}=\text{sf} P P' PP' & \\ = \text{bisimwImplies}\sqsubseteq\text{sf} P P' PP' & \text{,,} \\ \text{bisimwImplies}\sqsubseteq\text{sf}r P P' PP' & \end{aligned}$$

Finally we show as well that stable failure equivalence is as well implied by strong bisimilarity, using the fact that strong bisimilarity implies DRW-bisimilarity:

Theorem 10.10.2 (Agda Theorem)

$$\text{bisimslImplies}=\text{sf} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$$



$$\begin{aligned}
& (P : \text{Process} \infty \{lu\} c) \\
& (P' : \text{Process} \infty \{lu\} c) \\
& (PP' : \text{Bisims} \{\infty\} P P') \\
& \rightarrow P =_{\text{sf}} P'
\end{aligned}$$

Proof:

$$\begin{aligned}
& \text{bisimslImplies=sf } P P' PP' = \\
& \text{bisimwImplies=sf } P P' (\text{bisimsToBismw } P P' PP')
\end{aligned}$$

10.11 Bisimilarity Implies Failures Divergences Infinite Equivalence

As mentioned in Sect. 9, stable failures semantics doesn't deal well with divergent processes. Therefore the failures-divergences-infinite traces model was developed. When proving laws in this model, we face the same problems as for stable failures semantics, that the proofs are very evolved. As for stable failures semantics, the solution is to prove instead that strong and DRW-bisimilarity imply stable failures semantics, and prove the laws using bisimilarity. Therefore, in this section we show that DRW-bisimilarity and therefore as well strong bisimilarity imply equivalence with respect to *Failures/ Divergences/ Infinite Traces* in Agda.

FDI-refinement consists of four components: refinement with respect to trace semantics, failures, divergences, and with respect to infinite traces, and we will prove that DRW-bisimilarity implies each of these four refinements.

The first step towards this proof is that DRW-bisimilarity implies trace semantic, which was already defined in section 10.7.2.

The second step is to show that if P, P' are DRW-bisimilar, the failures for the process P imply the failures of the process P' . Note that we have in case of the FDI semantics two kind of failures: stable failures, and divergences, in which case a process can refuse everything because it is involved in performing infinitely many τ -transitions.

10.11.1 DRW-Bisimilarity Implies Refinement with respect to Failures

A proof that DRW-bisimilarity implies refinement with respect to failures is given in Agda as follows:



$\text{bisimRefusalros} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$
 $(P' : \text{Process} \infty \{lu\} c)$
 $(PP' : \text{Bisimw} \{\infty\} P P') (l : \text{List} (\text{Label } lu))$
 $(X : \text{Label } lu \rightarrow \text{Bool})$
 $(\text{fail} : \text{failure } P' l \text{ true } X)$
 $\rightarrow \text{failure } P l \text{ true } X$

$\text{bisimRefusalros } \{lu\} \{c\} P P' PP' l X$
 $(\text{stableFail } (\text{stableFp } Q' tr' stab' drefuse'))$
 $= (\text{stableFail } (\text{stableFp } Qhat trhat_2$
 $(\text{stabSchNoTickIfRos2StablePar } Qhat \text{ true } \text{stabSchQhat}$
 $\text{stabNoTick })$
 $\text{drefusehat}))$

where

$\text{Qcom} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1 P P' PP' l (\text{inj}_1 Q') tr'$

$\text{trcom} : \text{TrP } \{lu\} l (\text{bisimTraceTrP}_1 P P' PP' l$
 $(\text{inj}_1 Q') tr') P$
 $\text{trcom} = \text{bisimTraceTrP}_2 P P' PP' l (\text{inj}_1 Q') tr'$

$\text{QQ'com} : \text{BisimForNextP } (\text{bisimTraceTrP}_1 P P' PP' l$
 $(\text{inj}_1 Q') tr') (\text{inj}_1 Q')$
 $\text{QQ'com} = \text{bisimTraceTrP}_3 P P' PP' l (\text{inj}_1 Q') \quad tr'$

$Q : \text{Process} \infty \{lu\} c$
 $Q = \text{lemmayyy}_1 \text{Qcom } Q' stab' X drefuse' \text{QQ'com}$

$tr : \text{TrP } \{lu\} l (\text{inj}_1 Q) P$
 $tr = \text{lemmayyy}_2 \text{Qcom } l P Q' stab' X drefuse' \text{QQ'com } trcom$

$\text{QQ}' : \text{Bisimw } Q Q'$
 $\text{QQ}' = \text{lemmayyy}_3 \text{Qcom } Q' stab' X drefuse' \text{QQ'com}$

$\text{Qhat} : \text{Process} \infty \{lu\} c$
 $\text{Qhat} = \text{nonDivBecomeStable}_1 c Q$
 $(\text{bisimStableImpliesNotDivergent } c Q Q' \text{QQ}' stab'$
 $(\text{stableImpliesNonDiv } Q' stab'))$

$trhat : \text{TrP } \{lu\} [] (\text{inj}_1 \text{Qhat}) Q$
 $trhat = \text{nonDivBecomeStable}_2 c Q$





$$\begin{aligned}
 & (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ stab' \\
 & \quad (\text{stableImpliesNonDiv } Q' \ stab')) \\
 \\
 \text{QhatQ}' : & \quad \text{Bisimw Qhat } Q' \\
 \text{QhatQ}' = & \quad \text{bisimPPWithEmptyTr } Q \ Q' \ QQ' \ stab' \\
 & \quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ stab' \\
 & \quad (\text{stableImpliesNonDiv } Q' \ stab')) \ trhat \\
 \\
 \text{stabSchQhat} : & \quad \text{stableSch Qhat} \\
 \text{stabSchQhat} = & \quad \text{nonDivBecomeStable}_3 \ c \ Q \\
 & \quad (\text{bisimStableImpliesNotDivergent } c \ Q \ Q' \ QQ' \ stab' \\
 & \quad (\text{stableImpliesNonDiv } Q' \ stab')) \\
 \\
 \text{stabNoTick} : & \quad \text{noTickIfRoscoe true Qhat} \\
 \text{stabNoTick} = & \quad \text{bisimwStableToNoTick Qhat } Q' \ \text{QhatQ}' \ stab' \ \text{stabSchQhat} \\
 \\
 \text{trhat}_1 : & \quad \text{TrP } \{lu\} \ (l \ ++ \ []) \ (\text{inj}_1 \ \text{Qhat}) \ P \\
 \text{trhat}_1 = & \quad \text{trPAppendTrw } c \ P \ Q \ l \ [] \ (\text{inj}_1 \ \text{Qhat}) \ tr \ trhat \\
 \\
 \text{eqI} : & \quad (l \ ++ \ []) \equiv l \\
 \text{eqI} = & \quad \text{lemEqList } l \\
 \\
 \text{trhat}_2 : & \quad \text{TrP } \{lu\} \ l \ (\text{inj}_1 \ \text{Qhat}) \ P \\
 \text{trhat}_2 = & \quad \text{subst } (\lambda \ l' \rightarrow \text{TrP } \{lu\} \ l' \ (\text{inj}_1 \ \text{Qhat}) \ P) \ \text{eqI} \ \text{trhat}_1 \\
 \\
 \text{drefusehat} : & \quad \text{DRefusal Qhat true } X \\
 \text{drefusehat} = & \quad \text{bisimDRefusal } Q' \ \text{Qhat} \ stab' \\
 & \quad (\text{BismwSym Qhat } Q' \ \text{QhatQ}') \ X \ \text{true} \ drefuse' \\
 \\
 \text{bisimRefusalros } \{lu\}\{c\} \ P \ P' \ PP' \ l \ X & \\
 & \quad (\text{divergentFailure } (\text{trdiv } Q' \ trp' \ divq')) \\
 & \quad = (\text{divergentFailure } (\text{trdiv } Q \ tr \ divp)) \\
 \\
 \text{where} & \\
 \text{Qcom} : & \quad \text{Process } \infty \ \{lu\} \ c \uplus \text{ChoiceSet } c \\
 \text{Qcom} = & \quad \text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ trp' \\
 \\
 \text{trcom} : & \quad \text{TrP } \{lu\} \ l \ (\text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ trp') \ P \\
 \text{trcom} = & \quad \text{bisimTraceTrP}_2 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ trp' \\
 \\
 \text{QQ'com} : & \quad \text{BisimForNextP} \\
 & \quad (\text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ trp') \ (\text{inj}_1 \ Q')
 \end{aligned}$$


$$QQ'com = \text{bisimTraceTrP}_3 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \quad trp'$$

$$\begin{aligned} Q &: \text{Process} \infty \{lu\} \ c \\ Q &= \text{lemmaxxx}_1 \ Qcom \ Q' \ divq' \ QQ'com \end{aligned}$$

$$\begin{aligned} tr &: \text{TrP} \ \{lu\} \ l \ (\text{inj}_1 \ Q) \ P \\ tr &= \text{lemmaxxx}_2 \ Qcom \ l \ P \ Q' \ divq' \ QQ'com \ trcom \end{aligned}$$

$$\begin{aligned} QQ' &: \text{Bisimw} \ Q \ Q' \\ QQ' &= \text{lemmaxxx}_3 \ Qcom \ l \ Q' \ divq' \ QQ'com \end{aligned}$$

$$\begin{aligned} Q'Q &: \text{Bisimw} \ Q' \ Q \\ Q'Q &= \text{BismwSym} \ Q \ Q' \ QQ' \end{aligned}$$

$$\begin{aligned} divp &: \text{DivergentProcess} \infty \{lu\} \ c \ Q \\ divp &= \text{bisimImpliesDivergentPreserv} \ c \ Q' \ Q \ Q'Q \ divq' \end{aligned}$$

This implies refinement with respect to failures:

$$\begin{aligned} \text{bisimImFDI}_2 &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P : \text{Process} \infty \{lu\} \ c) \\ &\quad (P' : \text{Process} \infty \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\ &\quad \rightarrow P \sqsubseteq_{\text{fdi}_2\text{ros}} P' \end{aligned}$$

$$\text{bisimImFDI}_2 \ \{lu\} \{c\} \ P \ P' \ PP' = \text{bisimRefusalros} \ P \ P' \ PP'$$

$$\begin{aligned} \text{bisimImFDI}_{2r} &: \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P : \text{Process} \infty \{lu\} \ c) \\ &\quad (P' : \text{Process} \infty \{lu\} \ c) \\ &\quad (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\ &\quad \rightarrow P' \sqsubseteq_{\text{fdi}_2\text{ros}} P \end{aligned}$$

$$\text{bisimImFDI}_{2r} \ \{lu\} \{c\} \ P \ P' \ PP' = \text{bisimImFDI}_2 \ P' \ P \ (\text{BismwSym} \ P \ P' \ PP')$$

10.11.2 DRW-Bisimilarity Implies Refinement with respect to Divergences

We show that, if two processes P and P' are bisimilar, and P is a divergent process, then P' is divergent. This is proved in Agda as follows (we give only the proof for `Process`):

$$\text{bisimImpliesDivergentPreserv} : \{lu : \text{LUniv}\} (c : \text{Choice})$$

$$\begin{aligned}
& (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\
& (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\
& (divP : \text{DivergentProcess} \ \infty \ \{lu\} \ c \ P) \\
& \quad \rightarrow \text{DivergentProcess} \ \infty \ \{lu\} \ c \ P' \\
\text{bisimImpliesDivergentPreserv} \ c \ . (& \text{terminate} \ _) \ P' \ (\text{eqterminate} \ x) \ () \\
\text{bisimImpliesDivergentPreserv} \ c \ . (& \text{node} \ P) \ . (& \text{terminate} \ a) \ (\text{eqterminater} \ \{a\} \\
& \{ \text{node} \ P \} \ (\text{termeqnode} \ terequivP)) \ (\text{div} \ P \ \text{div}P) \\
& = \perp\text{-elim} \ (\text{divergentImpliesNotTermEquiv} \ + \ c \ P \ a \ terequivP \ \text{div}P) \\
\text{bisimImpliesDivergentPreserv} \ c \ . (& \text{node} \ P) \ . (& \text{node} \ P') \ (\text{eqnode} \ \{.P\} \ \{P'\} \ PP') \\
& (\text{div} \ P \ \text{div}P) \\
& = \text{div} \ P' \ (\text{bisimImpliesDivergentPreserv} \ + \ c \ P \ P' \ PP' \ \text{div}P)
\end{aligned}$$

Next we show that DRW bisimilarity implies that divergent traces are preserved:

$$\begin{aligned}
\text{bisimImTrD} : & \ \{lu : \text{LUniv}\} \ \{c : \text{Choice}\} \ (P : \text{Process} \ \infty \ \{lu\} \ c) \\
& (P' : \text{Process} \ \infty \ \{lu\} \ c) \\
& (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \ (l : \text{List} \ (\text{Label} \ lu)) \\
& (TrD : \text{TraceDivergent} \ \infty \ c \ l \ P') \\
& \quad \rightarrow \text{TraceDivergent} \ \infty \ c \ l \ P \\
\text{bisimImTrD} \ \{lu\} \ \{c\} \ P \ P' \ PP' \ l \ (\text{trdiv} \ Q' \ \text{trp}' \ \text{divp}') = & \text{trdiv} \ Q \ \text{tr} \ \text{divp} \\
\text{where} \\
Q\text{com} : & \ \text{Process} \ \infty \ \{lu\} \ c \ \uplus \ \text{ChoiceSet} \ c \\
Q\text{com} = & \text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \\
\text{trcom} : & \ \text{TrP} \ l \ (\text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \\
& \quad (\text{inj}_1 \ Q') \ \text{trp}') \ P \\
\text{trcom} = & \text{bisimTraceTrP}_2 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \\
QQ'\text{com} : & \ \text{BisimForNextP} \ (\text{bisimTraceTrP}_1 \ P \ P' \ PP' \ l \\
& \quad (\text{inj}_1 \ Q') \ \text{trp}') \ (\text{inj}_1 \ Q') \\
QQ'\text{com} = & \text{bisimTraceTrP}_3 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \\
Q : & \ \text{Process} \ \infty \ \{lu\} \ c \\
Q = & \text{lemmaxx}_1 \ Q\text{com} \ Q' \ \text{divp}' \ QQ'\text{com} \\
\text{tr} : & \ \text{TrP} \ l \ (\text{inj}_1 \ Q) \ P \\
\text{tr} = & \text{lemmaxx}_2 \ Q\text{com} \ l \ P \ Q' \ \text{divp}' \ QQ'\text{com} \ \text{trcom} \\
QQ' : & \ \text{Bisimw} \ Q \ Q' \\
QQ' = & \text{lemmaxx}_3 \ Q\text{com} \ l \ Q' \ \text{divp}' \ QQ'\text{com}
\end{aligned}$$

$Q'Q : \text{Bisimw } Q' Q$
 $Q'Q = \text{BismwSym } Q Q' QQ'$

 $\text{divp} : \text{DivergentProcess} \infty c Q$
 $\text{divp} = \text{bisimImpliesDivergentPreserv } c Q' Q Q'Q \text{ divp}'$

Therefore we get that DRW bisimilarity implies refinement with respect to divergences:

$\text{bisimImFDI}_1 : \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P P' : \text{Process+} \infty \{lu\} c)$
 $(PP' : \text{Bisimw+} \{\infty\} P P')$
 $\rightarrow P \sqsubseteq_{\text{fdi}_1} P'$
 $\text{bisimImFDI}_1 \{lu\}\{c\} P P' PP' = \text{bisimImTrD+ } P P' PP'$

 $\text{bisimImFDI}_{1r} : \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P P' : \text{Process+} \infty \{lu\} c)$
 $(PP' : \text{Bisimw+} \{\infty\} P P')$
 $\rightarrow P' \sqsubseteq_{\text{fdi}_1} P$
 $\text{bisimImFDI}_{1r} \{lu\}\{c\} P P' PP' = \text{bisimImFDI}_1 P' P (\text{BismwSym+ } P P' PP')$

10.11.3 DRW-Bisimilarity Implies Refinement with respect to Infinite Traces

We are going to show that if two processes are DRW-bisimilar, then any infinite trace of one is an infinite trace of the other. This is shown by reflecting each step of an infinite trace of one process by steps of the second process. Because of weak bisimulation, each step becomes a finite trace in the other process: a τ -transition can become arbitrarily many (or none) τ -transitions in the other, and an external choice becomes arbitrarily many τ -transitions, an external choice with same label followed by arbitrarily many τ -transitions.

One complication we have is that the definition of an infinite trace is a combined inductive/coinductive definition: only finitely many τ -transitions are allowed, but we have infinitely many external choices. Therefore the definition is inductive in the τ -transitions and coinductive in the external choices. So in the proof we need to take care about when we have an explicit external choice of the first process, in which case the size of the second process needs to go down when we reach the external choice – it will not go down in any initial extra τ -transitions.

The first step is to show that if we have an infinite trace starting with a process Q , and a finite trace without labels, leading up to it, we obtain an infinite trace extended by the new label. This is done in the following function `addTraceToInfiniteTrace`:

```

addTraceToInfiniteTrace : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))
  (tr1 : TrP [] (inj1 Q) P)
  (tr2 : infTr {i} l Q)
  → infTr {i} l P

addTraceToInfiniteTrace .(terminate x) .(terminate x) l (empty x) tr2 = tr2
addTraceToInfiniteTrace .(node P) Q l (tnode {·} [] {·} {P} tr1) tr2 =
  tnode (addTraceToInfiniteTrace+ P Q l tr1 tr2)

addTraceToInfiniteTrace+ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))
  (tr1 : TrP+ [] (inj1 Q) P)
  (tr2 : infTr {i} l Q)
  → infTr+ {i} l P

addTraceToInfiniteTrace+ P .(node P) l empty (tnode tr) = tr
addTraceToInfiniteTrace+ P Q l (intc .[] .(inj1 Q) x tr1) tr2 =
  intc l x (addTraceToInfiniteTrace (forceP (PI P x))
    Q l tr1 tr2)

addTraceToInfiniteTrace∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))
  (tr1 : TrP∞ [] (inj1 Q) P)
  (tr2 : infTr {i} l Q)
  → infTr∞ {↑ i} l P

forceP (addTraceToInfiniteTrace∞ P Q l tr1 tr2) =
  addTraceToInfiniteTrace (forceP P) Q l tr1 tr2

```

Now we show that infinite traces of one process become infinite traces of the other one. We present here only the most interesting case for `Process+`:

```

bisimInfTr+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (l : Stream (Label lu))
  (tr : infTr+ {i} l P')
  → infTr+ {i} l P
bisimInfTr+ {i} {lu} {c} P P' PP' l (extc .l e eq tr) =
  bisimInfTr+aux∞ P Q Q' l (Lab P' e) eq tr1 QQ' tr
where
  Q : Process∞ ∞ c
  Q = bisimEP'r PP' e

  Q' : Process∞ ∞ c
  Q' = PE P' e

  tr1 : TrP+ (Lab P' e :: []) (inj1 (forcep Q)) P
  tr1 = bisimEtrr PP' e

  eqlab : T' (Lab P' e ==| head l)
  eqlab = sym==| {lu} {head l} {Lab P' e} eq

  tr1' : TrP+ (head l :: []) (inj1 (forcep Q)) P
  tr1' = transLu {lu} (λ lab1 → TrP+ (lab1 :: []) (inj1
    (forcep Q)) P) eqlab tr1

  QQ' : Bisimw∞ Q Q'
  QQ' = bisimEnextr PP' e

```

This function was defined simultaneously with several auxiliary functions which deal with the first step made in a finite trace of the second process, where this finite trace is the reflection of a step of the infinite trace of the first process. We need to deal with it in this way in order to pass the termination checker:

```

bisimInfTr+aux∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  (Q : Process∞ ∞ c)
  (Q' : Process∞ ∞ c)
  (l : Stream (Label lu))
  (la : Label lu)
  (eqlab : T' (head l ==| la))
  (tr1 : TrP+ (la :: []) (inj1 (forcep Q)) P)

```

$$\begin{aligned}
& (QQ' : \text{Bisimw}\infty Q Q') \\
& (tr_2 : \text{infTr}\infty \{i\} (\text{tail } l) Q') \\
& \rightarrow \text{infTr+} \{i\} l P \\
\text{bisimInfTr+aux}\infty P Q Q' l . (\text{Lab } P x) \text{ eqlab} \\
& \quad (\text{extc } .[] .(\text{inj}_1 (\text{forcep } Q)) x tr_1) \\
& \quad QQ' tr_2 \\
& = \text{extc } l x \text{ eqlab} \\
& \quad (\text{addTraceToInfiniteTrace}\infty\infty (\text{PE } P x) Q (\text{tail } l) tr_1 \\
& \quad (\text{bisimInfTr}\infty Q Q' QQ' (\text{tail } l) tr_2)) \\
\\
\text{bisimInfTr+aux}\infty P Q Q' l la \text{ eqlab} \\
& \quad (\text{intc } .(la :: []) .(\text{inj}_1 (\text{forcep } Q)) x tr_1) QQ' tr_2 = \\
& \quad \text{intc } l x \\
& \quad (\text{bisimInfTr+aux}\infty\text{p} (\text{forcep } (\text{PI } P x)) Q Q' l la \text{ eqlab } tr_1 \\
& \quad QQ' tr_2) \\
\\
\text{bisim}\tau\text{InfTr+aux} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+} \infty \{lu\} c) \\
& \quad (Q : \text{Process} \infty c) \\
& \quad (Q' : \text{Process} \infty c) \\
& \quad (tr_1 : \text{TrP+} [] (\text{inj}_1 Q) P) \\
& \quad (QQ' : \text{Bisimw} Q Q') \\
& \quad (l : \text{Stream } (\text{Label } lu)) \\
& \quad (tr_2 : \text{infTr} \{i\} l Q') \\
& \rightarrow \text{infTr+} \{i\} l P \\
\text{bisim}\tau\text{InfTr+aux} P . (\text{node } P) . (\text{node } P') \\
& \quad \text{empty } (\text{eqnode } PP') l (\text{tnode } \{.l\} \{P'\} tr) = \text{bisimInfTr+} P P' PP' l tr \\
\\
\text{bisim}\tau\text{InfTr+aux} P Q Q' (\text{intc } .[] .(\text{inj}_1 Q) x tr_1) QQ' l tr_2 = \\
& \quad \text{intc } l x (\text{addTraceToInfiniteTrace} (\text{forcep } (\text{PI } P x)) Q l tr_1 \\
& \quad (\text{bisimInfTr} Q Q' QQ' l tr_2))
\end{aligned}$$

10.11.4 DRW-Bisimilarity Implies Refinement with respect to FDI

We have now all components ready, to prove that DRW-bisimilarity implies *Failures/Divergences/Infinite Traces* equivalence. First we combine the above proofs to one proof that DRW bisimilarity implies FDI refinement and equivalence:

$\text{bisimImFdiRef} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process} + \infty \{lu\} \ c)$
 $(PP' : \text{Bisimw} + \{\infty\} \ P \ P')$
 $\rightarrow P \sqsubseteq_{\text{fdi}} P'$
 $\text{bisimImFdiRef} \ P \ P' \ PP' = ((\text{bisimTraceEq} + P \ P' \ PP' \text{ ,,}$
 $\text{bisimImFDI}_1 \ P \ P' \ PP') \text{ ,,}$
 $\text{bisimImFDI}_2 + P \ P' \ PP') \text{ ,,}$
 $\text{bisimImFDI}_3 + P \ P' \ PP')$

Theorem 10.11.1 (Agda Theorem)

$\text{bisimImFdiEquiv} : \{lu : \text{LUniv}\} \{c : \text{Choice}\}$
 $(P \ P' : \text{Process} + \infty \{lu\} \ c)$
 $(PP' : \text{Bisimw} + \{\infty\} \ P \ P')$
 $\rightarrow P \equiv_{\text{fdi}} P'$

Proof:

$\text{bisimImFdiEquiv} \ P \ P' \ PP' = \text{bisimImFdiRef} \ P \ P' \ PP' \text{ ,,}$
 $\text{bisimImFdiRef} \ P' \ P \ (\text{BismwSym} + P \ P' \ PP')$

Using that strong bisimilarity implies DRW-bisimilarity, we obtain a proof that strong bisimilarity implies as well FDI refinement and equivalence:

$\text{bisimslmFdiRef} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process} + \infty \{lu\} \ c)$
 $(PP' : \text{Bisims} + \{\infty\} \ P \ P')$
 $\rightarrow P \sqsubseteq_{\text{fdi}} P'$

$\text{bisimslmFdiRef} \ P \ P' \ PP' = \text{bisimImFdiRef} \ P \ P' (\text{bisimsToBismw} + P \ P' \ PP')$

$\text{bisimslmFdiEquiv} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process} + \infty \{lu\} \ c)$
 $(PP' : \text{Bisims} + \{\infty\} \ P \ P')$
 $\rightarrow P \equiv_{\text{fdi}} P'$

$\text{bisimslmFdiEquiv} \ P \ P' \ PP' = \text{bisimImFdiEquiv} \ P \ P'$
 $(\text{bisimsToBismw} + P \ P' \ PP')$

10.12 Proofs in Divergence-Respecting Weak Bisimilarity Semantics

A number of techniques have been developed to prove algebraic laws for CSP in this thesis; the first one was a direct proof of algebraic laws using the traces model. Trace semantics is one of the most common models for determining safety properties.

As we discussed in 7, trace semantics refers only to the observable traces. It does not distinguish between external and internal choice. In particular, it does not tell what a process can refuse to do.

The stable failures model records the events that a process performs with a set of events a process fails to perform after a process stabilises. The stable failures model is not effective in analysing processes which can diverge, which means they have an infinite sequence of τ -transitions. The stable failures model ignores any divergent behaviour.

In the *Failures/Divergences/Infinite Traces* (FDI) model of CSP, these behaviours are introduced alongside failures information. In this approach, we can identify a process P with the failures/divergences/infinite traces that may be observed.

As we noted before, proofs in the stable failures and FDI model are very difficult, since we have to prove all properties not only for the initial process, but as well for all subprocesses evolving from the initial processes. It is as well very tedious to prove all the different components (2 in case of stable failures, and 4 in case of FDI) of the semantic models. It is much easier to prove these laws with respect to DRW-bisimilarity, which implies equivalence with respect to the traces, the stable failures and the failures/divergences/infinite traces models.

10.12.1 Proof of Commutativity of the External Choice Operator

We prove commutativity of the external choice operator with respect to the three main semantic models, by showing that $(P \sqcap Q)$ and $(Q \sqcap P)$ are strongly bisimilar. The fact that we have strong bisimilarity in this case (which doesn't hold for many algebraic laws) makes the proof easier, proofs of DRW-bisimilarity require much more work. As for proofs of trace equivalence, we need to apply the function `fmap` to one part of the equation, in order to adjust the return values of termination events.

Theorem 10.12.1 (Agda Theorem)

$\text{C}\square++ : \{lu : \text{LUniv}\} \{c_0\ c_1 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c_0)$

$$\begin{aligned}
& (Q : \text{Process} + \infty \{lu\} c_1) \\
& \rightarrow \text{Bisims} + (P \sqbox ++ Q) (\text{fmap} + \text{swap} \sqcup (Q \sqbox ++ P))
\end{aligned}$$

Proof:

$$\begin{aligned}
\text{bisim2E} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2E} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisimELab} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{refl} \\
\text{bisimELab} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{refl} \\
\text{bisimENext} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \\
& \quad \text{lemBisimFmap}\infty \text{inj}_2 \text{swap} \sqcup (\text{PE } P x) \\
\text{bisimENext} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \\
& \quad \text{lemBisimFmap}\infty \text{inj}_1 \text{swap} \sqcup (\text{PE } Q y) \\
\text{bisim2I} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2I} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisimINext} \quad & (\text{C}\sqbox ++ \{lu\} P Q) (\text{inj}_1 x) = \\
& \quad \text{C}\sqbox \infty ++ \{lu = lu\} (\text{PI } P x) Q \\
\text{bisimINext} \quad & (\text{C}\sqbox ++ \{lu\} P Q) (\text{inj}_2 y) = \\
& \quad \text{C}\sqbox + \infty + \{lu = lu\} P (\text{PI } Q y) \\
\text{bisim2T} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2T} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisim2TEq} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{refl} \\
\text{bisim2TEq} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{refl} \\
\text{bisim2Er} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2Er} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisimELabr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{refl} \\
\text{bisimELabr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{refl} \\
\text{bisimENextr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \\
& \quad \text{lemBisimFmap}\infty \text{inj}_1 \text{swap} \sqcup (\text{PE } Q x) \\
\text{bisimENextr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \\
& \quad \text{lemBisimFmap}\infty \text{inj}_2 \text{swap} \sqcup (\text{PE } P y) \\
\text{bisim2I}r \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2I}r \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisimINextr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \\
& \quad \text{C}\sqbox + \infty + P (\text{PI } Q x) \\
\text{bisimINextr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \\
& \quad \text{C}\sqbox \infty ++ (\text{PI } P y) Q \\
\text{bisim2Tr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{inj}_2 x \\
\text{bisim2Tr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{inj}_1 y \\
\text{bisim2TEqr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_1 x) = \text{refl} \\
\text{bisim2TEqr} \quad & (\text{C}\sqbox ++ P Q) (\text{inj}_2 y) = \text{refl}
\end{aligned}$$

Now we can use the fact that strong bisimilarity implies DRW-weak bisimilarity, trace equivalence, stable-failure equivalence, and FDI-equivalence, and obtain the laws in all these four semantics as well:

Theorem 10.12.2 (Agda Theorem)

$\text{SW}\square+ : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\}$
 $(P : \text{Process}+ \infty \{lu\}\ c_0)$
 $(Q : \text{Process}+ \infty \{lu\}\ c_1)$
 $\rightarrow \text{Bisimw}+ (P \square++ Q) (\text{fmap}+ \text{swap}\sqcup (Q \square++ P))$

Proof:

$\text{SW}\square+ P\ Q = \text{bisimsToBisimw}+ (P \square++ Q)$
 $(\text{fmap}+ \text{swap}\sqcup (Q \square++ P)) (\text{C}\square++ P\ Q)$

Theorem 10.12.3 (Agda Theorem)

$\text{commuteExtChTrace}+ : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\}$
 $(P : \text{Process}+ \infty \{lu\}\ c_0)$
 $(P' : \text{Process}+ \infty \{lu\}\ c_1)$
 $\rightarrow (P \square++ P') \equiv+ (\text{fmap}+ \text{swap}\sqcup (P' \square++ P))$

Proof:

$\text{commuteExtChTrace}+ P\ P' = \text{bisimTraceEqs}+= (P \square++ P')$
 $(\text{fmap}+ \text{swap}\sqcup (P' \square++ P))$
 $(\text{C}\square++ P\ P')$

Theorem 10.12.4 (Agda Theorem)

$\text{commuteExtChSF}+ : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\}$
 $(P : \text{Process}+ \infty \{lu\}\ c_0)$
 $(P' : \text{Process}+ \infty \{lu\}\ c_1)$
 $\rightarrow (P \square++ P') =\text{sf}+ (\text{fmap}+ \text{swap}\sqcup (P' \square++ P))$

Proof:

$\text{commuteExtChSF}+ P\ P' = \text{bisimsImplies}=\text{sf}+ (P \square++ P')$
 $(\text{fmap}+ \text{swap}\sqcup (P' \square++ P))$
 $(\text{C}\square++ P\ P')$

Theorem 10.12.5 (Agda Theorem)

$$\begin{aligned} \text{commuteExtChFDI+} : & \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\ & (P : \text{Process+} \ \infty \ \{lu\} \ c_0) \\ & (P' : \text{Process+} \ \infty \ \{lu\} \ c_1) \\ & \rightarrow (P \ \square++ \ P') \equiv \text{fdi+} \ (\text{fmap+} \ \text{swap} \ \sqcup \ (P' \ \square++ \ P)) \end{aligned}$$
Proof:

$$\begin{aligned} \text{commuteExtChFDI+} \ P \ P' = & \text{bisimSImFdiEquiv} \ (P \ \square++ \ P') \\ & (\text{fmap+} \ \text{swap} \ \sqcup \ (P' \ \square++ \ P)) \\ & (\text{C} \ \square++ \ P \ P') \end{aligned}$$
10.12.2 Proof of Commutativity of the Interleaving Operator

The proof in this section uses the same steps as the one in the previous section. We start by showing that the process $(P \ ||| \ Q)$ is strongly bisimilar to $(Q \ ||| \ P)$. The proof in Agda as follows:

Theorem 10.12.6 (Agda Theorem)

$$\begin{aligned} \text{C}|||+ : & \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\ & (P : \text{Process+} \ \infty \ \{lu\} \ c_0) \\ & (Q : \text{Process+} \ \infty \ \{lu\} \ c_1) \\ & \rightarrow \text{Bisims+} \ (P \ |||++ \ Q) \ (\text{fmap+} \ \text{swap} \ \times \ (Q \ |||++ \ P)) \end{aligned}$$
Proof:

$$\begin{aligned} \text{bisim2E} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{inj}_2 \ x \\ \text{bisim2E} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_2 \ y) = \text{inj}_1 \ y \\ \text{bisimELab} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{refl} \\ \text{bisimELab} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_2 \ y) = \text{refl} \\ \text{bisimENext} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{C}||| \ \infty + \ (\text{PE} \ P \ x) \ Q \\ \text{bisimENext} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_2 \ y) = \text{C}||| \ \infty + \ P \ (\text{PE} \ Q \ y) \\ \text{bisim2I} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{inj}_2 \ x \\ \text{bisim2I} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_2 \ y) = \text{inj}_1 \ y \\ \text{bisimINext} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{C}||| \ \infty + \ (\text{PI} \ P \ x) \ Q \\ \text{bisimINext} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_2 \ y) = \text{C}||| \ \infty + \ P \ (\text{PI} \ Q \ y) \\ \text{bisim2T} \quad & (\text{C}|||+ \ P \ Q) \ (x \ \text{,,} \ x_1) = (x_1 \ \text{,,} \ x) \\ \text{bisim2TEq} \quad & (\text{C}|||+ \ P \ Q) \ (x \ \text{,,} \ x_1) = \text{refl} \\ \text{bisim2Er} \quad & (\text{C}|||+ \ P \ Q) \ (\text{inj}_1 \ x) = \text{inj}_2 \ x \end{aligned}$$

We obtain again proofs of these laws using DRW-weak bisimilarity, trace equivalence, stable-failure equivalence, and FDI-equivalence:

Proof:

Proof:

$$\text{commute} \parallel \text{Trace} + P P' = \text{bisimTraceEqs} + (P \parallel ++ P') \\ (\text{fmap} + \text{swap} \times (P' \parallel ++ P)) \\ (\text{C} \parallel + P P')$$

Theorem 10.12.9 (Agda Theorem)

$$\begin{aligned} \text{commute}||\text{SF}+ & : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\} \\ & (P : \text{Process}+ \infty \{lu\}\ c_0) \\ & (P' : \text{Process}+ \infty \{lu\}\ c_1) \\ & \rightarrow (P |||++ P') =_{\text{sf}+} (\text{fmap}+ \text{swap} \times (P' |||++ P)) \end{aligned}$$
Proof:

$$\begin{aligned} \text{commute}||\text{SF}+ P P' &= \text{bisimslmplies}=\text{sf}+ (P |||++ P') \\ & \quad (\text{fmap}+ \text{swap} \times (P' |||++ P)) \\ & \quad (\text{C}|||++ P P') \end{aligned}$$
Theorem 10.12.10 (Agda Theorem)

$$\begin{aligned} \text{commute}||\text{FDI}+ & : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\} \\ & (P : \text{Process}+ \infty \{lu\}\ c_0) \\ & (P' : \text{Process}+ \infty \{lu\}\ c_1) \\ & \rightarrow (P |||++ P') \equiv_{\text{fdi}+} (\text{fmap}+ \text{swap} \times (P' |||++ P)) \end{aligned}$$
Proof:

$$\begin{aligned} \text{commute}||\text{FDI}+ P P' &= \text{bisimslmFdiEquiv} (P |||++ P') \\ & \quad (\text{fmap}+ \text{swap} \times (P' |||++ P)) \\ & \quad (\text{C}|||++ P P') \end{aligned}$$

10.13 Proof of the Monadic Laws

10.13.1 Proof of First Monadic Law

We defined processes in a monadic way, and will in this section prove two monad laws for processes.

In functional programming, a monad is given by a functor \mathbf{M} together with morphisms

$$\gg= : \mathbf{M}\ A \rightarrow (A \rightarrow \mathbf{M}\ B) \rightarrow \mathbf{M}\ B$$

and

$$\text{return} : A \rightarrow \mathbf{M}\ A$$

such that the following laws hold:

$$\begin{aligned}
\text{return } a \gg= f &= f \ a \\
p \gg= \text{return} &= p \\
(p \gg= f) \gg= g &= p \gg= (\lambda x. f \ x \gg= g)
\end{aligned}$$

The proof of the first monadic law is trivial since

$$(\text{terminate } a \gg= P)$$

is definitionally equal to P . As before, first we prove that $(\text{terminate } a \gg= P)$ bisimilar to P . We can directly prove both strong and weak bisimilarity using reflexivity of this relation:

Theorem 10.13.1 (Agda Theorem)

$$\begin{aligned}
\text{monadicLaw}_1s : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\
(a : \text{ChoiceSet } c_0) \\
(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\
\rightarrow \text{Bisims } (\text{terminate } a \gg= P) (P \ a)
\end{aligned}$$

Proof:

$$\text{monadicLaw}_1s \ a \ P = \text{BismsRef } (P \ a)$$

Theorem 10.13.2 (Agda Theorem)

$$\begin{aligned}
\text{monadicLaw}_1 : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\
(a : \text{ChoiceSet } c_0) \\
(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\
\rightarrow \text{Bisimw } (\text{terminate } a \gg= P) (P \ a)
\end{aligned}$$

Proof: $\text{monadicLaw}_1 \ a \ P = \text{BismwRef } (P \ a)$

We obtain now that the first monadic law holds with respect to trace equivalence, stable failures, and FDI equivalence as follows:

Theorem 10.13.3 (Agda Theorem)

$$\begin{aligned}
\text{monadicLaw}_1\text{Trace} : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \\
(a : \text{ChoiceSet } c_0) \\
(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} \ c_1) \\
\rightarrow (\text{terminate } a \gg= P) \equiv \text{tr } (P \ a)
\end{aligned}$$

Proof: $\text{monadicLaw}_1\text{Trace} \ a \ P = \text{bisimTraceEq=}$

$$(\text{terminate } a \gg= P) (P \ a) (\text{monadicLaw}_1 \ a \ P)$$

Theorem 10.13.4 (Agda Theorem)

$\text{monadicLaw}_1\text{SF}+ : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\}$
 $(a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1)$
 $\rightarrow (\text{terminate } a \gg= P) =_{\text{sf}} (P\ a)$

Proof:

$\text{monadicLaw}_1\text{SF}+ \ a\ P = \text{bisimwImplies}=\text{sf}$
 $(\text{terminate } a \gg= P) (P\ a) (\text{monadicLaw}_1\ a\ P)$

Theorem 10.13.5 (Agda Theorem)

$\text{monadicLaw}_1\text{FDI}+ : \{lu : \text{LUniv}\}\{c_0\ c_1 : \text{Choice}\}$
 $(a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1)$
 $\rightarrow (\text{terminate } a \gg= P) \equiv_{\text{fdi}} (P\ a)$

Proof:

$\text{monadicLaw}_1\text{FDI}+ \ a\ P = \text{bisimFDIImpEq}$
 $(\text{terminate } a \gg= P) (P\ a) (\text{monadicLaw}_1\ a\ P)$

10.13.2 Proof of Third Monadic Law

Next, we carry out the proof of the third monadic law:

$$(P \gg=+ (Q \gg=+ R))$$

is equal to

$$((P \gg=+ Q) \gg=+ R)$$

with respect to our semantic models.

As before, we first prove that the two processes are strongly bisimilar. The proof in CSP-Agda is as follows:

Theorem 10.13.6 (Agda Theorem)

$\text{monadicLaw}_{1-3}\infty : \{lu : \text{LUniv}\}\{c_0\ c_1\ c_2 : \text{Choice}\}$

$$\begin{aligned}
& (P : \text{Process} \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\
& \rightarrow \text{Bisims} \infty ((P \gg=\infty Q) \gg=\infty R) \\
& \quad (P \gg=\infty (\lambda x \rightarrow Q x \gg=\infty R))
\end{aligned}$$

Proof:

$$\text{forceB } (\text{monadicLaw}_{1-3} \infty P Q R) = \text{monadicLaw}_{1-3} (\text{forceP } P) Q R$$

$$\begin{aligned}
\text{monadicLaw}_{1-3} : & \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\
& (P : \text{Process} \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\
& \rightarrow \text{Bisims} ((P \gg= Q) \gg= R) \\
& \quad (P \gg= (\lambda x \rightarrow Q x \gg= R)) \\
\text{monadicLaw}_{1-3} & (\text{terminate } x) Q R = \\
& \quad \text{BisimsRef } (((\text{terminate } x \gg= Q) \gg= R)) \\
\text{monadicLaw}_{1-3} & (\text{node } x) Q R = \text{eqnode } (\text{monadicLaw}_{1-3+} x Q R)
\end{aligned}$$

$$\begin{aligned}
\text{monadicLaw}_{1-3+} : & \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\
& (P : \text{Process}+ \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\
& \rightarrow \text{Bisims}+ ((P \gg=+ Q) \gg=+ R) \\
& \quad (P \gg=+ (\lambda x \rightarrow Q x \gg= R))
\end{aligned}$$

$$\begin{aligned}
\text{bisim2E} & (\text{monadicLaw}_{1-3+} P Q R) e = e \\
\text{bisimELab} & (\text{monadicLaw}_{1-3+} P Q R) e = \text{refl} \\
\text{bisimENext} & (\text{monadicLaw}_{1-3+} P Q R) e = \\
& \quad \text{monadicLaw}_{1-3} \infty (\text{PE } P e) Q R \\
\text{bisim2I} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_1 (\text{inj}_1 x)) = \text{inj}_1 x \\
\text{bisim2I} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_1 (\text{inj}_2 y)) = \text{inj}_2 y \\
\text{bisim2I} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_2 ()) \\
\text{bisimINext} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_1 (\text{inj}_1 x)) = \\
& \quad \text{monadicLaw}_{1-3} \infty (\text{PI } P x) Q R \\
\text{bisimINext} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_1 (\text{inj}_2 y)) = \\
& \quad \text{monadPT}+ P Q R y \\
\text{bisimINext} & (\text{monadicLaw}_{1-3+} P Q R) (\text{inj}_2 ()) \\
\text{bisim2T} & (\text{monadicLaw}_{1-3+} P Q R) () \\
\text{bisim2TEq} & (\text{monadicLaw}_{1-3+} P Q R) () \\
\text{bisim2Er} & (\text{monadicLaw}_{1-3+} P Q R) e = e
\end{aligned}$$

```

bisimELabr (monadicLaw1-3+ P Q R) e = refl
bisimENextr (monadicLaw1-3+ P Q R) e = monadicLaw∞ P Q R e
bisim2lr    (monadicLaw1-3+ P Q R) (inj1 x) = inj1 (inj1 x)
bisim2lr    (monadicLaw1-3+ P Q R) (inj2 y) = inj1 (inj2 y)
bisimlNextr (monadicLaw1-3+ P Q R) (inj1 x) = monadicLaw1-3∞ (PI P x) Q R
bisimlNextr (monadicLaw1-3+ P Q R) (inj2 y) =
    monadPT+ P Q R y
bisim2Tr    (monadicLaw1-3+ P Q R) e = e
bisim2TEqr  (monadicLaw1-3+ P Q R) ()

```

By using the proof that strong bisimilar implies DRW- bisimilar we prove that the processes are DRW-bisimilar:

Theorem 10.13.7 (Agda Theorem)

```

monad3SW+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → Bisimw+ ((P >>=+ Q) >>=+ R)
    (P >>=+ (λ x → Q x >>= R))

```

Proof:

```

monad3SW+ P Q R = bisimsToBismw+
  ((P >>=+ Q) >>=+ R)
  (P >>=+ (λ x → Q x >>= R))
  (monadicLaw1-3+ P Q R)

```

Now we obtain proofs that the two processes are equivalent with respect to traces, stable failures and FDI semantics:

Theorem 10.13.8 (Agda Theorem)

```

monadicLaw3Trace+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>=+ Q) >>=+ R) ≡ tr+
    (P >>=+ (λ x → Q x >>= R))

```

Proof:

```

monadicLaw3Trace+ P Q R = bisimTraceEqs+=

```

$$\begin{aligned}
& ((P \gg=+ Q) \gg=+ R) \\
& (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\
& (\text{monadicLaw}_{1-3+} P Q R)
\end{aligned}$$

Theorem 10.13.9 (Agda Theorem)

$$\begin{aligned}
\text{monadicLaw}_3\text{SF+} : & \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\
& (P : \text{Process+} \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\
\rightarrow & ((P \gg=+ Q) \gg=+ R) = \text{sf+} \\
& (P \gg=+ (\lambda x \rightarrow Q x \gg= R))
\end{aligned}$$

Proof:

$$\begin{aligned}
\text{monadicLaw}_3\text{SF+ } P Q R = & \text{bisimlImplies=sf+} \\
& ((P \gg=+ Q) \gg=+ R) \\
& (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\
& (\text{monadicLaw}_{1-3+} P Q R)
\end{aligned}$$

Theorem 10.13.10 (Agda Theorem)

$$\begin{aligned}
\text{monadicLaw}_3\text{FDI+} : & \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\
& (P : \text{Process+} \infty \{lu\} c_0) \\
& (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\
& (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\
\rightarrow & ((P \gg=+ Q) \gg=+ R) \equiv \text{fdi+} \\
& (P \gg=+ (\lambda x \rightarrow Q x \gg= R))
\end{aligned}$$

Proof:

$$\begin{aligned}
\text{monadicLaw}_3\text{FDI+ } P Q R = & \text{bisimlImFdiEquiv} \\
& ((P \gg=+ Q) \gg=+ R) \\
& (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\
& (\text{monadicLaw}_{1-3+} P Q R)
\end{aligned}$$

10.13.3 The Second Monadic Law

The reader might wonder what happens to the second monadic law which says that

$$P \gg= \text{terminate} \text{ is equal to } P$$

It turns out that the two processes are, at least with the current definition of $_ \gg = _$, in general not DRW-bisimilar: Let $R := l \longrightarrow \text{STOP}$ for some label l , i.e. R has as only event an l -transition to STOP . Consider a process P having a \checkmark -event with return type a and a τ -transition to R , and let $Q := P \gg = \text{terminate}$. Process Q has a τ -transition to $(\text{terminate } a)$ and a τ -transition to R . What P can do, Q can do as well: the \checkmark -event can be simulated by a τ -transition to $(\text{terminate } a)$, and the τ -transition to R by the same τ -transition for Q . However, Q can have a τ -transition to $(\text{terminate } a)$, and this cannot be simulated by P , since P cannot reach a state which is bisimilar to $(\text{terminate } a)$.

Actually in Schneider's version of stability, the two processes are not equal with respect to stable failures semantics, since Q has a stable state $(\text{terminate } a)$ in which it refuses everything, whereas P reaches with τ -transitions only process R , which cannot refuse l . According to Roscoe's version of stability, at least according to our interpretation ($(\text{terminate } a)$ is not a CSP process), $(\text{terminate } a)$ is not stable, therefore the processes are equal with respect to stable failures semantics and with respect to FDI semantics à la Roscoe. One might argue that even with Roscoe $(\text{terminate } a)$ should be a stable state. Since this has strong implications we leave it for future work to explore this.

One might argue that a τ -transition to $(\text{terminate } a)$ should be treated similar to a \checkmark -transition with return value a . However, they behave differently. Let P_0 have a τ -transition to $(\text{terminate } a)$, and an l' -transition to STOP . Let P_1 having a \checkmark -event with return value a and the same l' -transition to STOP . Process $P_0 \parallel R$ can have a τ transition to $(\text{terminate } a) \parallel R$, a state in which it can refuse l' . Process $P_1 \parallel R$ cannot execute the \checkmark -transition, since it needs to synchronise with a \checkmark -transition for R . It is stable, and cannot refuse l' .

Possible solution. One solution one might consider is to change the definition of $P \gg = Q$, so that \checkmark -events of P with return value a become, in case $(Q \ a)$ is a terminated process, termination events. In order to do this, we need to replace the type of $Q \ a$ from $\text{Process}\infty$ to Process . We give here a definition of the revised monadic bind: We first define a function deciding whether a process has terminated:

```
isTerminate : {i : Size}{lu : LUniv} {c : Choice}(P : Process i {lu} c)
              → Bool
isTerminate (terminate x) = true
isTerminate (node x)      = false
```

```

isNode : {i : Size}{lu : LUniv} {c : Choice}(P : Process i {lu} c)
        → Bool
isNode = ¬b ∘ isTerminate

```

We can extract for terminated process the return value:

```

processIsTerminateToResult : {i : Size}{lu : LUniv} {c : Choice}
                             (P : Process i {lu} c)
                             (isTer : True (isTerminate P))
                             → ChoiceSet c
processIsTerminateToResult (terminate x) isTer = x
processIsTerminateToResult (node x) ()

```

Now the definition of the revised monadic bind is as follows:

```

_>>=Str_ : {c0 : Choice} → String
           → (ChoiceSet c0 → String) → String
s >>=Str f = s ++s ">>" ++s choice2Str2Str f

```

mutual

```

_>>=∞_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
         → Process∞ i {lu} c0
         → (ChoiceSet c0 → Process i {lu} c1)
         → Process∞ i {lu} c1
forcep (P >>=∞ Q) = forcep P >>= Q
Str∞ (P >>=∞ Q) = Str∞ P >>=Str (Str ∘ Q)

```

```

_>>=_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
        → Process i c0
        → (ChoiceSet c0 → Process i {lu} c1)
        → Process i c1
node P >>= Q = node (P >>=+ Q)
terminate x >>= Q = Q x

```

```

_>>=+_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
         → Process+ i c0
         → (ChoiceSet c0 → Process i {lu} c1)
         → Process+ i c1
E (P >>=+ Q) = E P
Lab (P >>=+ Q) = Lab P

```

$$\begin{aligned}
\text{PE } (P \gg=+ Q) \ c &= \text{PE } P \ c \gg= \infty Q \\
\text{I } (P \gg=+ Q) &= \text{I } P \uplus' \text{subset}' (\text{T } P) \\
&\quad (\text{isNode} \circ (Q \circ (\text{PT } P))) \\
\text{PI } (P \gg=+ Q) (\text{inj}_1 \ c) &= \text{PI } P \ c \gg= \infty Q \\
\text{forcep } (\text{PI } (P \gg=+ Q) (\text{inj}_2 (\text{sub } a \ x))) &= Q (\text{PT } P \ a) \\
\text{Str} \infty (\text{PI } (P \gg=+ Q) (\text{inj}_2 (\text{sub } a \ x))) &= \text{Str } (Q (\text{PT } P \ a)) \\
\text{T } (P \gg=+ Q) &= \text{subset}' (\text{T } P) \\
&\quad (\text{isTerminate} \circ (Q \circ (\text{PT } P))) \\
\text{PT } (P \gg=+ Q) (\text{sub } a \ x) &= \text{processIsTerminateToResult } (Q (\text{PT } P \ a)) \ x \\
\text{Str}+ (P \gg=+ Q) &= \text{Str}+ P \gg= \text{Str } (\text{Str} \circ Q)
\end{aligned}$$

The problem with this definition is however, that it doesn't respect DRW-bisimilarity with respect to the second argument: $Q := \lambda a \rightarrow \text{terminate } a$ and $Q' := \lambda a \rightarrow (\tau \rightarrow \text{terminate } a)$ are extensionally weakly bisimilar, but $P \gg= Q$ and $P \gg= Q'$ behave quite differently. In order to fix it one would need to decide whether $Q \ a$ is termination equivalent, which is undecidable.

One solution is to accept the fact that the 2nd monad law doesn't hold. Having the first and third monad law is a structure with quite good properties, since they correspond very well to the fact that monadic bind means first executing the first process or program, and then depending on it executing the second one. That the second argument one is actually an always terminating process or program might not happen very often, so it might not be a big problem of not having the second monadic law.

Chapter 11

Case Study: Safety for Railway Interlocking Systems

Systems, for which failure or malfunction cannot be tolerated, and for which failure may result in one or more hazardous outcomes like loss of life, significant property damage or damage to the environment, are called critical systems (Knight [2002], Storey [1996], Fowler [2009]). There are three kinds of critical systems:

- **Safety critical systems**, where the failure of the system could result in loss of life or damage to the environment (Storey [1996]). Examples are medical devices, e.g. automated infusion pumps; systems in the area of aerospace, such as civil aviation, military aviation, and manned space travel; and traffic control, such as Railway control system, air traffic control, road traffic control, and automotive control systems.
- **Mission critical systems**, where a malfunction may fail some goal-directed activity, for example, a navigational system of a space probe (Storey [1996]).
- **Business critical systems**, where failure could lead to very high costs for the business using that system. Examples are the customer account system in a bank, an online shopping cart, areas where secrecy is required such as secret service, and sensitive areas in companies.

Formal methods can be used in the design and development of such systems to reduce the risk of failure (Rushby [1989]). CSP-Agda aims to support the modelling and verification of critical systems such as railways systems.

In this chapter we demonstrate our results by modelling a simple scenario for railways in CSP and CSP-Agda. We will investigate the use of the process algebra CSP together with the model checker FDR and of CSP-Agda to verify the system. Checking our model against the given safety properties, the signalling principles, will provide, in case of failure, counterexamples that



help to debug the given possible scenario design. We then prove deadlock and livelock freedom of the corrected system. This work gives a successful example of how CSP-Agda can be used to support the industrial development process.

11.1 Specifying an Possible Scenario for Railways in Natural Language

In modern railway traffic, various train services usually run on the same track of a railway section (Huisman and Boucherie [2001]). Railway signalling system are used to direct train traffic and keep trains clear of each other at all times. Trains move on a fixed path, so, unless they derail, the only reason for collision is that two trains are allowed to access the same train segment (Newman [1995]). In the following scenario for railways, two trains leave different stations heading toward each other, to cross the same segment with a single track. The signalling system should guarantee the safe movement for both trains. Therefore, the train asks the signal control system for being allowed to cross the segment. If the segment is free (not occupied by another train), then the signalling system will set the signal to green, otherwise, it will keep it red, see the following Fig. 11.1.

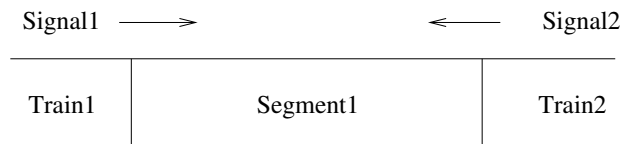


Figure 11.1: Text

11.2 CSP and CSP-Agda Specification

We consider a railway as consisting of the three separate physical components shown in Fig. 11.1.

- The signals, which can show the aspects green or red according to the status of the segment.
- The train segments.
- Finally, the train component. In our scenario, we have two trains.



We will in the following model the processes in CSP by using CSP M, which is the machine-readable derivation of CSP introduced in Roscoe [1998].

We have only one segment, therefore the type of segments is given as follows

```
datatype SEGMENT = seg1
```

This is specified in CSP-Agda in the following way:

```
data SEGMENT : Set where seg1 : SEGMENT
```

We have one signal on each end of the segment, guarding access to the segment, specified in CSP and CSP-Agda in the following way:

```
datatype SIGNAL = sig1 | sig2
```

```
data SIGNAL : Set where sig1 sig2 : SIGNAL
```

In our case, the set TRAIN contains two elements only. This setup is defined in CSP and CSP-Agda as follows:

```
datatype TRAIN = ta | tb
```

```
data TRAIN : Set where ta tb : TRAIN
```

In this definition, we named the trains by `ta` and `tb`. The signal aspects in our settings can be either red or green, specified in CSP and CSP-Agda in the following way:

```
datatype ASPECT = red | green
```

```
data ASPECT : Set where red green : ASPECT
```

The segment can be in two states, either free, which means there is no train in the segment or blocked, which means the segment currently occupied by a train. This is defined in CSP and CSP-Agda as follows:

```
datatype SEG_STATE = free | blocked
```

```
data SEGSTATE : Set where free blocked : SEGSTATE
```

We also statically define some events or channels. The first one is `get_seg`, which allows a train to request for a segment to be in a state, which can be either free or blocked. This is specified in CSP in the following way:

channel get_seg : TRAIN.SEGMENT.SEG_STATE <

In Agda we define a data type of labels and constructors for each channel. The definition in CSP-Agda for the type of labels together with the constructor corresponding to get_seg is as follows:

```
data LabelTrains : Set where
  getSegm : TRAIN → SEGMENT → SEGSTATE → LabelTrains
```

We have the channel set_seg, which allows a train to request that a segment is set to a given state. This is described in CSP as follows:

channel set_seg : TRAIN.SEGMENT.SEG_STATE

In CSP-Agda we add a constructor to LabelTrains:

```
setSegm : TRAIN → SEGMENT → SEGSTATE → LabelTrains
```

Similarly, set_sig allows a train to request a signal to be set to an aspect, which can be red or green. This is defined in CSP as the following channel and in CSP-Agda as a constructor of LabelTrains:

channel set_sig : TRAIN.SIGNAL.ASPECT

```
setSig : TRAIN → SIGNAL → ASPECT → LabelTrains
```

Finally, we define the event, which sets a signal to a given aspect in CSP and CSP-Agda:

channel set_sigs : SIGNAL.ASPECT

```
setSigs : SIGNAL → ASPECT → LabelTrains
```

Next, we specify the dynamic part of the scenario for railways. Using CSP language, we model the behaviour of a possible scenario for railways through several processes that interact with each other: the Train, the Signals, and the Segment.

First we introduce the notion of external choice indexed over an index set. In CSP M notation

$$(\parallel \text{ newasp} : \text{ASPECT} @ \text{set_sig.t.sig.newasp} - > \text{SIG_CTL}(\text{sig}))$$

stands for the external choice of

$$\text{set_sig.t.sig.newasp} \rightarrow \text{SIG_CTL}(\text{sig})$$

indexed over newasp : ASPECT.

In CSP-Agda we define the function $|\Box|$, which defines the external choice of processes indexed over a list of elements of type A . So it takes as arguments a list of elements of type A , and a function, mapping elements of A to processes, and produces as result the external choice over those processes: It is defined as follows:

$$\begin{aligned} |\Box| &: \{i : \text{Size}\} \{c : \text{Choice}\} \{lu : \text{LUniv}\} \{A : \text{Set}\} \rightarrow \text{List } A \\ &\quad \rightarrow (A \rightarrow \text{Process } i \{lu\} c) \rightarrow \text{Process } i \{lu\} c \\ |\Box| \{i\} \{c\} [] f &= \text{STOP } c \\ |\Box| \{i\} \{c\} (a :: []) f &= f a \\ |\Box| \{i\} \{c\} (a :: (b :: l)) f &= \text{fmap } c \uplus c \rightarrow c ((f a) \Box ((|\Box| (b :: l) f))) \end{aligned}$$

The movement of a train is constrained by the current aspect of the signal. The control system for signals sets a signal to a new aspect repeatedly. In CSP we obtain the following definition:

$$\text{SIG_CTL}(\text{sig}) = ([\text{t} : \text{TRAIN} \text{ @ } [\text{newasp} : \text{ASPECT} \text{ @ } \text{set_sig.t.sig.newasp} \rightarrow \text{SIG_CTL}(\text{sig})])$$

In CSP-Agda the definition is as follows:

$$\begin{aligned} \text{SIGCTL} &: \{i : \text{Size}\}(\text{sig} : \text{SIGNAL}) \rightarrow \text{Process}_{\infty} i \{\text{labelTrains}\} \emptyset' \\ \text{forcep}(\text{SIGCTL } \text{sig}) &= |\Box| \{lu = \text{labelTrains}\} \text{LabelListTRAIN } \lambda tr \rightarrow \\ &\quad |\Box| \{lu = \text{labelTrains}\} \text{LabelListASPECT } \lambda asp \rightarrow \\ &\quad \text{lab}(\text{setSig } tr \text{ sig } asp) \longrightarrow \text{SIGCTL } \text{sig} \\ \text{Str}_{\infty}(\text{SIGCTL } \text{sig}) &= \text{"SIGCTL"} ++ \text{showSIGNAL } \text{sig} \end{aligned}$$

Here \longrightarrow is the prefix operation in CSP-Agda, where the process argument is an element of Process_{∞} .

The process $\text{SEG_CTL}(\text{seg}, \text{segstate})$ for monitoring the segment can get the state of the segment, and set its state. It is defined in CSP as follows:

```

SEG_CTL(seg,segstate) =
  ([ t : TRAIN @
    get_seg.m.t.seg.segstate - > SEG_CTL(seg,segstate))
  ([ ])
  ([ t : TRAIN @ ([ newsegstate : SEG_STATE @
    set_seg.m.t.seg.newsegstate - > SEG_CTL(seg,newsegstate)))

```

In CSP we first define the two subprocesses for getting and setting the segment, and then define the external choice of these two processes. Note that this is as before a corecursive definition, so we define it by guarded recursion, for which the fact that the corecursive call is called after applying the observation **forcep** to the corecursively defined function. We require as well the operation **fmap**, which in this case changes the return type $\emptyset \uplus \emptyset$ to \emptyset – this doesn't do anything but is needed for type correctness.

mutual

```

SEGCTL1 : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
  → Process $\infty$  i {labelTrains}  $\emptyset'$ 
forcep (SEGCTL1 seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
  lab (getSegm tr seg segstate)
  → SEGCTL seg segstate
Str $\infty$  (SEGCTL1 seg segstate) = "SEGCTL1" ++s showSEGMENT seg
  ++s showSEGSTATE segstate

SEGCTL2 : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
  → Process $\infty$  i {labelTrains}  $\emptyset'$ 
forcep (SEGCTL2 seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
  | $\square$ |tr LabelListSEGSTATE  $\lambda$  newsegstate →
  lab (setSegm tr seg newsegstate)
  → SEGCTL seg newsegstate
Str $\infty$  (SEGCTL2 seg segstate) = "SEGCTL2" ++s showSEGMENT seg
  ++s showSEGSTATE segstate

SEGCTL : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
  → Process $\infty$  i {labelTrains}  $\emptyset'$ 
forcep (SEGCTL seg segstate) = fmap {labelTrains}  $\emptyset \uplus \emptyset \rightarrow \emptyset$ 
  ( forcep (SEGCTL1 seg segstate)  $\square$ wNam
    forcep (SEGCTL2 seg segstate)
    Using  $\square$ toStringSimple ,  $\square$ fmapNameSimple ,

```

```

Str∞ (SEGCTL seg segstate) = "SEGCTL" ++s showSEGMENT seg
                                ++s showSEGSTATE segstate
                                □fmapNameSimple)

```

Here `□wNam_Using_,_,_` is external choice, but with explicit functions for forming the string name of the resulting process depending on the names for its process arguments. This is used so that CSP-Agda-Simulator displays a good name for this process. The function has 3 arguments: one forming the name if we have external choice, which depends on the names for the two processes; one for forming the name in case the process has become `fmap` applied to the first process, which depends on the name for that process; and one for forming it if it has become `fmap` applied to the second process, depending on the name for the second process. Its type is

```

□wNam_Using_,_,_ : {c0 c1 : Choice} → {i : Size} → {lu : LUniv}
  → Process i {lu} c0
  → Process i {lu} c1
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)
  → (□fmapRightName : String → String)
  → Process i {lu} (c0 ⊕' c1)

```

The processes train enter `TRAIN_ENTER(tr,segm,sig)` and train leave `TRAIN_LEAVE(tr,segm,sig)` model the functionality of moving the train from and to the segment in question. `TRAIN_ENTER` checks whether the segment is free, then sets the signal to green and then the segment to being blocked, and then switches to the `TRAIN_LEAVE` process. The `TRAIN_LEAVE` process sets the segment to free, sets the signal to red and switches back to the `TRAIN_ENTER`. The reader might discover that this is unsafe – this is deliberate and will be discussed later. The definition is as follows:

```

TRAIN_ENTER(tr,segm,sig) = get_segm.tr.segm.free
                          - > set_sig.tr.sig.green
                          - > set_segm.tr.segm.blocked
                          - > TRAIN_LEAVE(tr,segm,sig)
TRAIN_LEAVE(tr,segm,sig) = set_segm.tr.segm.free
                          - > set_sig.tr.sig.red
                          - > TRAIN_ENTER(tr,segm,sig)

```

The definition in CSP-Agda is as follows:

```

TRAINENTER : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
  → Process∞ i {labelTrains} ∅'
forcep (TRAINENTER tr seg sig) = lab (getSegm tr seg free)
  →pp ((lab (setSig tr sig green)
  →pp (lab (setSegm tr seg blocked)
  → TRAINLEAVE tr seg sig)))
Str∞ (TRAINENTER tr seg sig) = "TRAINENTER" ++s showTRAIN tr ++s
  showSEGMENT seg ++s showSIGNAL sig

TRAINLEAVE : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
  → Process∞ i {labelTrains} ∅'
forcep (TRAINLEAVE tr seg sig) = lab (setSegm tr seg free)
  →pp (lab (setSig tr sig red)
  → TRAINENTER tr seg sig)
Str∞ (TRAINLEAVE tr seg sig) = "TRAINLEAVE" ++s showTRAIN tr ++s
  showSEGMENT seg ++s showSIGNAL sig

```

Here \rightarrow_{pp} is the prefix operator but with argument in `Process`.

Finally, we have three controllers, which control the signals in each part of the segment and the status of a segment. Each one works independently from the other, and we use the interleaving operator to represent it in CSP. This combination is defined as follows:

$$(\text{SIG_CTL}(\text{sig1}) \parallel \text{SIG_CTL}(\text{sig2}) \parallel \text{SEG_CTL}(\text{seg1}, \text{free}))$$

In CSP-Agda it is defined as follows:

```

SYSTEMp1 : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅' ×' ∅')
SYSTEMp1 = (SIGCTL sig1 |||wNam∞ SIGCTL sig2
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple)
  |||wNam∞ SEGCTL seg1 free
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple

```

Here we were using $_|||wNam__Using______$, which is interleaving $_|||______$, but has similar to $_|||wNam__Using______$ extra arguments for forming a good name for the resulting process from the names for its arguments.

In case of train movement, we have two controllers, which control the movement of the train from each side of the segment. These controllers work as well independently, so we use the interleaving operator to define it in CSP:

$(\text{TRAIN_ENTER}(\text{ta}, \text{seg1}, \text{sig1}) \parallel \parallel \text{TRAIN_ENTER}(\text{tb}, \text{seg1}, \text{sig2}))$

The definition in CSP-Agda as follows:

```
SYSTEMp2 : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅')
SYSTEMp2 = TRAINENTER ta seg1 sig1 |||wNam∞
          TRAINENTER tb seg1 sig2 Using
          |||toStringSimple , fmapNameSimple , fmapNameSimple
```

Based on the combined controllers process we form a larger process, called System, in which trains, segment and signals interact by synchronising all events. The complete system is defined in CSP as follows:

$$\begin{aligned} \text{SYSTEM} = & (\text{SIG_CTL}(\text{sig1}) \parallel \parallel \text{SIG_CTL}(\text{sig2}) \parallel \parallel \text{SEG_CTL}(\text{seg1}, \text{free})) \\ & [[\{ | \text{get_segm}, \text{set_segm}, \text{set_sig} | \}]] \\ & (\text{TRAIN_ENTER}(\text{ta}, \text{seg1}, \text{sig1}) \parallel \parallel \text{TRAIN_ENTER}(\text{tb}, \text{seg1}, \text{sig2})) \end{aligned}$$

It is defined in CSP-Agda as follows:

```
SYSTEM : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
SYSTEM = SYSTEMp1 [ (λ x → true) ] |||wNam∞ [ (λ x → true) ] SYSTEMp2
          Using |||toStringSimple , fmapNameSimple , fmapNameSimple
```

11.3 Verification of a Possible Scenario for Railways Using FDR

The basic safety property we want to prove for our layout is there are no two green signal at the same time. We formalise this principle regarding signals. This provides a higher-level description than a formalisation using trains and segments, since it directly expresses the property that never both signals are green at the same time. This is important since we can only verify that a process fulfils its specification, but not that the specification is sufficient to guarantee the requirement of safety – the latter is an instance of validation. Therefore, it is crucial that it is as clear as possible that the formal specification really codifies the desired requirement.

Once we formalised the properties appropriately, then we can apply to our model. We model these general principles as CSP processes. The FDR tool then allows us to check the refinement between the properties and the primary model.

The general idea of modelling properties is that we model a good behaviour that does satisfy the principle and show a given process refines it, or a wrong behaviour, which we want to exclude it from the model, and show that a given process does not refine it.

In the following, we describe more concretely how this works for our properties. In our setting, as we discuss it, in the beginning, we want to check our system is free from the harmful behaviour like, that there are two green signals at the same time. We formalise this property in CSP as follows:

$$\begin{aligned} \text{BAD_SIGNALS} = & \text{set_sigs.sig1.green} - > \text{set_sigs.sig2.green} - > \text{STOP} \\ & ([]) \\ & \text{set_sigs.sig2.green} - > \text{set_sigs.sig1.green} - > \text{STOP} \end{aligned}$$

Now we abstract our system by hiding the `set_seg`, `get_seg` channels using the hiding operator and then renaming the channel `set_sig` to `set_sigs`. Therefore only the change of the signals is visible. We defined it in CSP as follows:

$$\begin{aligned} \text{SYSTEM_SIGONLY} = & \\ & (\text{SYSTEM } \{ | \text{set_seg}, \text{get_seg} | \}) \\ & [[\text{set_sig.ta.sig1.red} < - \text{set_sigs.sig1.red} , \\ & \quad \text{set_sig.ta.sig1.green} < - \text{set_sigs.sig1.green}, \\ & \quad \text{set_sig.ta.sig2.red} < - \text{set_sigs.sig2.red}, \\ & \quad \text{set_sig.ta.sig2.green} < - \text{set_sigs.sig2.green}, \\ & \quad \text{set_sig.tb.sig1.red} < - \text{set_sigs.sig1.red} , \\ & \quad \text{set_sig.tb.sig1.green} < - \text{set_sigs.sig1.green}, \\ & \quad \text{set_sig.tb.sig2.red} < - \text{set_sigs.sig2.red}, \\ & \quad \text{set_sig.tb.sig2.green} < - \text{set_sigs.sig2.green}]] \end{aligned}$$

In CSP-Agda we first carry out the hiding:

```
hideInSystem : Label labelTrains → Bool
hideInSystem (lab (setSig _ _)) = false
hideInSystem (lab (setSigs _ _)) = false
hideInSystem l = true
```

```
SYSTEMSHIDE : {i : Size} → Process∞ i {labelTrains}
```

$$(\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset'))$$

`SYSTEMSHIDE` = `HideWithName` ∞ `nameHidelnSystem` `hidelnSystem` `SYSTEM`

and then the renaming:

$$\begin{aligned} \text{renamelnSystem} &: \text{Label labelTrains} \rightarrow \text{Label labelTrains} \\ \text{renamelnSystem} (\text{lab} (\text{setSig } x \ x_1 \ x_2)) &= \text{lab} (\text{setSigs } x_1 \ x_2) \\ \text{renamelnSystem } l &= l \end{aligned}$$

$$\begin{aligned} \text{SYSTEM-SIGONLY} &: \{i : \text{Size}\} \rightarrow \text{Process} \infty i \{\text{labelTrains}\} \\ &\quad (\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')) \\ \text{SYSTEM-SIGONLY} &= \text{RenameWithName} \infty \text{nameRenamlnSystem} \\ &\quad \text{renamelnSystem SYSTEMSHIDE} \end{aligned}$$

Now we can check whether our system has that bad behaviour by using the FDR tools. To this end, we use the following CSP formula expressing that `SYSTEM-SIGONLY` refines with respect to trace semantics `BAD-SIGNALS`:

$$\text{assert SYSTEM-SIGONLY [T= BAD-SIGNALS]}$$

When we used the FDR tools, see Fig.11.2, we find, as expected, that the bad trace occurs in the system, which means that it is possible to have two signals green at the same time and that a collision could happen. The reason is a race condition: train 1 can check whether the segment is free, and then set the signal to green. Before it then sets the segment to blocked, train 2 can check whether the segment is free, which is still the case, since it hasn't been set to be blocked. It can then set the other signal to green. Then both trains set the segment to be blocked.

Race conditions arise in programs, which have multiple threads. For more information about race condition can be found in Netzer and Miller [1992], Emrath et al. [1989], and Nudler and Rudolph [1986].

11.4 Verification/Simulation of the Case Study

In this section, we use different simulator tools for CSP to simulate our possible scenario for railways. Among these tools, we use the ProBE tools and CSP-Agda-Simulator.

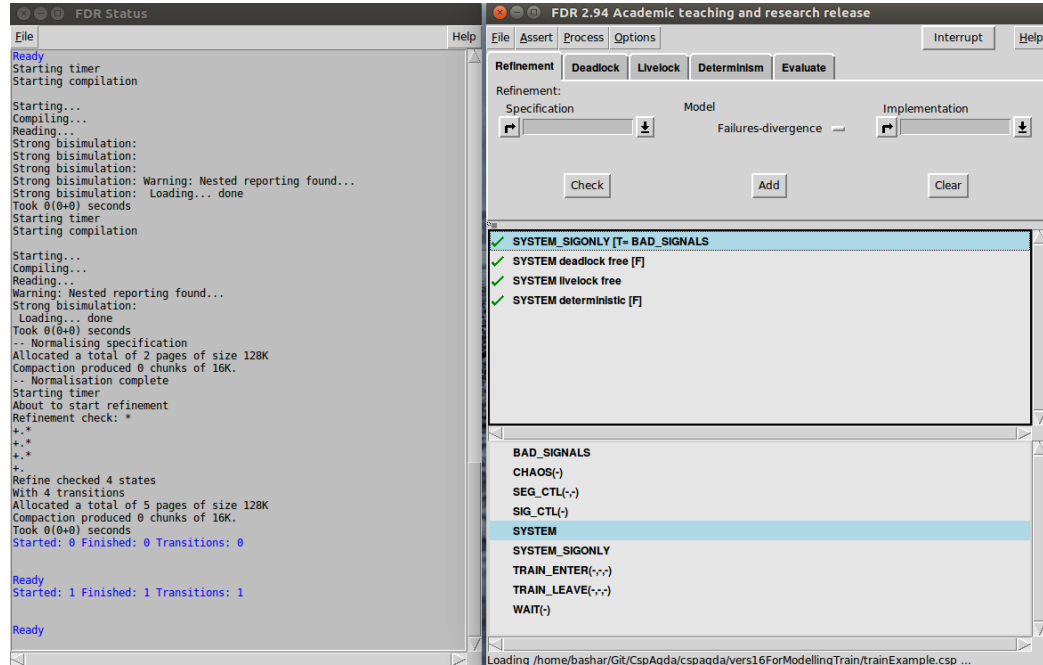


Figure 11.2: Bad behaviour Using FDR Tools

11.4.1 Simulate using ProBE

In CSP, the ProBE tool is used to simulate specifications. The user interface presents a state and all events that are possible from that state. The user may then continue through the model along every possible valid trace by selecting the events of that trace. The user is prevented from selecting any events which are not part of the trace. For instance, if a system has only one external choice, with say label a , then the user will not be allowed to engage in an event with label b . The ProBE is helpful in checking for the existence of traces, which the user does not wish to be valid in the specification. In our case, we simulate the bad behaviour using the ProBE tool as shown in Fig.11.3

11.4.2 Simulation using CSP-Agda-Simulator

Similar to the ProBE tools, our simulator in CSP-Agda has the same possibilities, see Chapter 6 for more information. An example run of the simulator is shown in Fig. 11.4

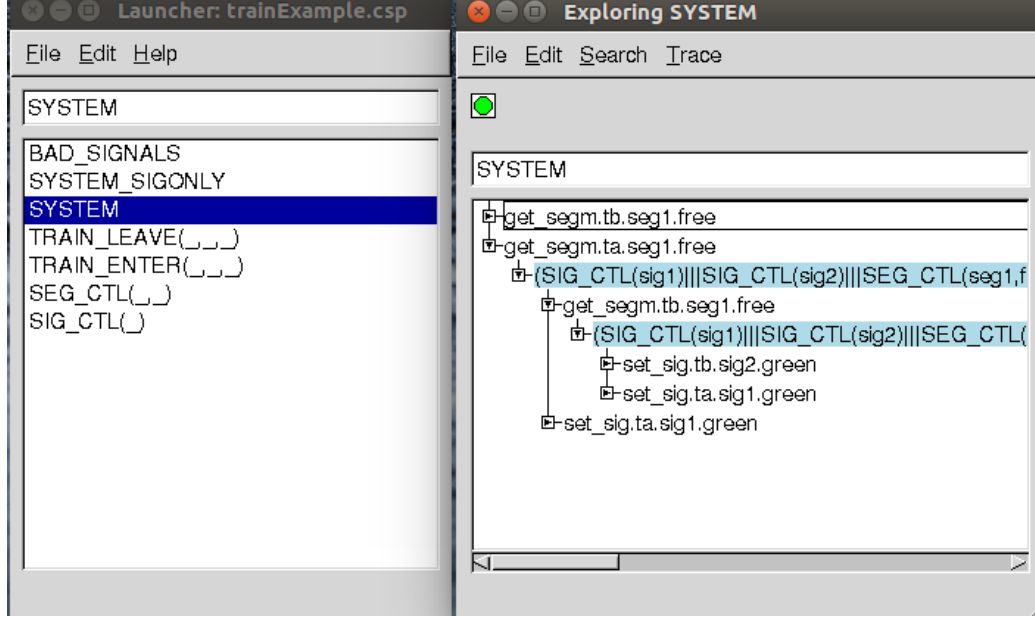


Figure 11.3: Simulate bad behaviour Using ProBE Tools

11.4.3 Verification in CSP-Agda

In CSP-Agda we can show the refinement statement. In order to simplify the proof we first optimise the system slightly, by replacing the choice sets so that the decode to a finite set of n elements `Fin n`. This reduces substantially the number of cases. The following theorem is shown by making case distinctions on possible traces for `BAD_SIGNALS`, and showing, that each trace is a trace of `OPTIMIZED-SYSTEM`. Here The CSP-Agda simulator is helpful, because with it we can determine the exact trace needed and which choices are to be made. The proof is as follows (we omit the boring case distinction in the proof, the full proof can be found in the appendix):

```
badSignal : OPTIMIZED-SYSTEM  $\sqsubseteq_{\infty}$  BAD_SIGNALS
badSignal = -- Proof Omitted in thesis,
            -- available in the CSP-Agda repository
```

11.5 Correcting the System

In order to fix the problem we change the process `SEG_CTL`: When asked for whether the segment is in a state, it switches to a new process `SEG_CTL1`, in which state it will accept only requests for the segment in question to be set to a new state, after which it returns to the original process. This means

```

> ./agda/trainExample
Termination-Events:
Events: :τ int(r.r.(r.l.r.0,r.0)):τ
Choose Event
int(r.r.(r.l.l.0,l.0))
-τ->
Termination-Events:
Events: ext(r.(l.l.l.r.0,l.0)):sig1green int(r.r.(r.l.r.0,r.0)):τ
Choose Event
ext(r.(l.l.l.r.0,l.0))
-sig1green->
Termination-Events:
Events: int(r.r.(r.l.r.0,r.0)):τ int(r.r.(r.r.l.r.0,l.0)):τ
Choose Event
int(r.r.(r.l.r.0,r.0))
-τ->
Termination-Events:
Events: ext(r.(l.r.r.r.0,r.0)):sig2green int(r.r.(r.r.l.r.0,l.0)):τ
Choose Event
ext(r.(l.r.r.r.0,r.0))
-sig2green->
Termination-Events:
Events: int(r.r.(r.r.l.r.0,l.0)):τ int(r.r.(r.r.r.r.0,r.0)):τ
Choose Event

```

Figure 11.4: Simulate Bad behaviour Using CSP-Agda Simulator

that this process essentially takes a lock on controlling the segment, which can only be released by setting the state of the segment in question to a new state.

The definition in CSP is as follows:

```

SEG_CTL(seg,segstate) =
  ([ t : TRAIN @
    get_seg.m.t(seg,segstate) - > SEG_CTL1(seg,segstate))
  ([ )
  ([ t : TRAIN @ ([ newsegstate : SEG_STATE @
    set_seg.m.t(seg,newsegstate) - > SEG_CTL(seg,newsegstate)))

SEG_CTL1(seg,segstate) =
  ([ t : TRAIN @ ([ newsegstate : SEG_STATE @
    set_seg.m.t(seg,newsegstate) - > SEG_CTL(seg,newsegstate))

```

The definition in CSP-Agda is as follows:

mutual

SEGCTLa : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)

```

    → Process∞ i {labelTrains} ∅'
forcep (SEGCTLa seg segstate) = |□|tr LabelListTRAIN λ tr →
                                lab (getSegm tr seg segstate)
                                → SEGCTL1 seg segstate
Str∞ (SEGCTLa seg segstate) = "SEGCTLa" ++s showSEGMENT seg
                              ++s showSEGSTATE segstate

SEGCTLb : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process∞ i {labelTrains} ∅'
forcep (SEGCTLb seg segstate) = |□|tr LabelListTRAIN      λ tr →
                                |□|tr LabelListSEGSTATE λ newsegstate →
                                lab (setSegm tr seg newsegstate)
                                → SEGCTL seg newsegstate
Str∞ (SEGCTLb seg segstate) = "SEGCTLb" ++s showSEGMENT seg
                              ++s showSEGSTATE segstate

SEGCTL1 : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process∞ i {labelTrains} ∅'
forcep (SEGCTL1 seg segstate) = |□|tr LabelListTRAIN λ tr →
                                |□|tr LabelListSEGSTATE λ newsegstate →
                                lab (setSegm tr seg newsegstate)
                                → SEGCTL seg newsegstate
Str∞ (SEGCTL1 seg segstate) = "SEGCTL1" ++s showSEGMENT seg
                              ++s showSEGSTATE segstate

SEGCTL : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process∞ i {labelTrains} ∅'
forcep (SEGCTL seg segstate) = fmap ∅⊕∅→∅
                              ( forcep (SEGCTLa seg segstate) □wNam
                                forcep (SEGCTLb seg segstate)
                                Using □toStringSimple , □fmapNameSimple ,
                                □fmapNameSimple)
Str∞ (SEGCTL seg segstate) = "SEGCTL" ++s showSEGMENT seg
                              ++s showSEGSTATE segstate

```

A second correction needed is that in the TRAIN_LEAVE process we need to set first the signal to red and then the segment to free. Otherwise, when the segment is set to free, the other train could see that its free and set it to blocked and its signal to green, resulting in two green signals. The

corrected version is as follows

```

TRAIN_ENTER(tr,segm,sig) = get_segm.tr.segm.free
                           - > set_sig.tr.sig.green
                           - > set_segm.tr.segm.blocked
                           - > TRAIN_LEAVE(tr,segm,sig)
TRAIN_LEAVE(tr,segm,sig) = set_sig.tr.sig.red
                           - > set_segm.tr.segm.free
                           - > TRAIN_ENTER(tr,segm,sig)

```

Now the ProBE tool rejects the refinement statement, and the violating trace cannot be found any more in the simulator of CSP-Agda.

We can simulate the process using the CSP-Agda simulator. However, that we don't get a bad trace requires manually to simulate all possibilities, until and future traces would be too long for the bad trace, and checks that the bad trace is not a possibility.

What is more convincing is to prove in CSP-Agda, that CSP-Agda does not refine the bad system. What one can do is determine a trace for **BAD SIGNALS**, and then show that this trace is not a trace of **OPTIMIZED-SYSTEM**. It results in a rather big case distinction, in which we expand all possible proofs that it is a trace of **OPTIMIZED-SYSTEM**, and show that none is actually a proof. This can be done purely mechanically: The code (we exclude the very long and boring case distinction) is as follows:

```

badTraceLabels : List (Label labelTrains)
badTraceLabels = lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: []

badTraceBadSignal : Tr $\infty$  {labelTrains} badTraceLabels nothing BAD SIGNALS
badTraceBadSignal = tnode (extc (lab (setSigs sig2 green) :: []) nothing (inj1 zero)
                             (tnode (extc [] nothing zero
                                     (tnode empty))))))

noTraceBadSignal : (m : Maybe (ChoiceSet ( $\emptyset'$   $\times'$   $\emptyset'$   $\times'$   $\emptyset'$   $\times'$  ( $\emptyset'$   $\times'$   $\emptyset'$ ))))
                  (l : List (Label labelTrains))
                  (tr : Tr $\infty$  {labelTrains} l m OPTIMIZED-SYSTEM)
                  (mm' : m  $\equiv$  nothing)
                  (ll' : l  $\equiv$  badTraceLabels)
                   $\rightarrow \perp$ 

noTraceBadSignal = -- Proof Omitted in thesis,

```

-- available in the CSP-Agda repository

```
badSignalProof : ¬ (OPTIMIZED-SYSTEM  $\sqsubseteq_{\infty}$  BADSIGNALS)
badSignalProof x = noTraceBadSignal nothing badTraceLabels
  (x badTraceLabels nothing badTraceBadSignal)
  refl refl
```

It turns out that even such a small example is quite time consuming, and requires a lot of computer resources for type checking it. In the future we want to improve the optimisation, so that at least such kind of proofs can be done easily.

In fact, such kind of finite examples should be delegated to a model checker. Karim Kanso has developed a plugin which allows to integrate automated theorem proving tools into Agda (Kanso [2012], Kanso and Setzer [2014]). Unfortunately, at the time of writing the thesis it is no longer supported, although some researchers have become interested in supporting it. It would still require substantial work to combine it with CSP-Agda, but it would be very beneficial: we could do finite boring proofs using a model checker, and concentrate on proving interactively more universal statements, such as relationships between CSP-Agda proofs and specifications, which one cannot prove using a model checker, which requires a fixed finite system.



Chapter 12

Summary

This research aims to give the type theoretic interactive theorem prover Agda the ability to model and verify concurrent programs by representing the process algebra CSP in monadic form, together with its semantics. In our approach, we represent processes coinductively. The termination checker of Agda guarantees productivity of processes. This allows defining processes recursively without having to reduce them to the recursion combinator. Since processes are given coinductively, we introduce processes by extended primitive corecursion, also called guarded recursion. The principle of primitive corecursion guarantees that processes are productive. This means that for a process we can determine whether it terminates or not. In case it terminates, we can compute the result returned, and, in case it doesn't terminate, we can determine, which next transitions it can make, and the next processes after firing these transitions.

Processes in our approach are similar to interactive programs. They are defined using an atomic operation, corresponding to the next transitions they can make. The operators of CSP are in our approach defined operations, which combine processes defined from atomic operations. The set of processes forms a monad (Process A), which depends on a set A. Using this we can define a dependent composition (monadic bind) and a dependent loop construct `rec` for processes.

We have written a simulator, called CSP-Agda-Simulator in Agda. The simulator displays the process as a string. Then it computes and displays the set of events and their results, and of external and internal choices together with their labels. The simulator is written in the same language Agda as the language in which proofs are carried out, using the unique feature of Agda of being both a dependently typed programming language and an interactive theorem prover.

We implemented trace semantics of CSP in Agda, together with the corresponding refinement and equality relations, formally in CSP-Agda. To

demonstrate the proof capabilities of CSP-Agda, we proved in CSP-Agda selected algebraic laws of CSP based on trace semantics. In our approach, while processes are defined inductively, trace semantics is defined inductively, using the fact that traces are finite objects.

We have implemented the stable failures model and the failures, divergences and infinite traces (FDI) model of CSP in Agda, together with the corresponding refinement and equality relations.

We extended the library CSP-Agda by implementing strong bisimilarity and divergent-respecting weak (DRW) bisimilarity in CSP-Agda. We have shown that processes bisimilar with respect to one of these two notions are trace equivalent, stable failures equivalent and FDI equivalent. We have as well shown that strong bisimilarity implies DRW bisimilarity.

This makes it easier to develop proofs of algebraic properties for trace, stable failures, and FDI semantics, by showing that they are strongly or DRW bisimilar. As an example, we applied this methodology to prove algebraic laws according to this semantics.

As a case study, we modelled a scenario from the railway domain in CSP and used FDR tools to check this model is free from deadlock, livelock, and to prove refinement statements. We used the ProBE tool and CSP-Agda-Simulator to simulate the model. We showed how to carry out proofs of refinement in CSP-Agda, although it would be more suitable to integrate model checkers into CSP-Agda in order to carry out such kind of proofs.

We have therefore developed a library which allows to represent CSP processes, simulate them, together with their main semantics, and which allows to prove properties including proofs of algebraic laws with respect to the different semantics.

Chapter 13

Future Work

We have developed elements of the European Rail Traffic Management System ERTMS (ERTMS [2013]) in CSP, and we plan to implement those processes in CSP-Agda, and to prove safety, liveness, and refinement statements in CSP-Agda. This will require automated theorem proving techniques in order to carry out larger case studies. Here we could use Kanso's PhD thesis Kanso [2012] (see as well Kanso and Setzer [2014]), in which he verified real-world railway interlocking systems in Agda. Verifying larger examples might require upgrading the integration of SAT solvers and model checkers into Agda2, which has been developed by Kanso [2012], to the current version of Agda, and to extend that work.

What needs to be investigated, whether the integration of SAT solvers and model checkers into Agda by Kanso could be used to prove properties about CSP processes, especially in finite situations such as the processes occurring in chapter11 in a better way. What needs to be explored is whether under certain conditions one can translate a refinement statement in CSP-Agda, we want to prove, into a model checking property. Then it might be possible to prove this model checking problem using a model checker and obtain the refinement statement in CSP-Agda directly. Proofs by hand in CSP-Agda could then be confined to generic problems (quantifying over arbitrary processes), whereas problems of finite fixed processes could be solved automatically using model checking. We would obtain a library, which would allow to prove general algebraic properties, which cannot be shown by model checkers, interactively, while proving specific properties of finite processes using model checkers automatically. A direct solution would be to integrate the CSP model checker FDR2 directly into Agda.

The simple case study carried out showed that more optimisations need to be carried out for CSP-processes so that at least simple examples can be carried out by hand. Part of it is due to the fact that the efficiency of Agda's type checker has room for improvements. In addition, we plan

to investigate, whether processes can be more optimised or the data type of processes improved, so that such simple proofs can be carried out easily by hand in a more transparent way. For instance it might be possible to automatically determine traces up to a number of steps, together with a correctness proofs, and use this to prove refinement in the various semantics of CSP.

We have the vision to write prototypes of programs, e.g. of some elements of the ERTMS, in Agda and make them directly executable in Agda. For this, a major step for CSP-Agda, namely to be able to program directly with CSP processes in Agda, needs to be set up. Then we could use the fact that Agda is both a theorem prover and a dependently typed programming language, to have programs written and their correctness proofs in the same language, without the need to translate between different languages, and therefore the need to verify the correctness of such a translation.

More proofs of algebraic laws need to be carried out as future work, in order to demonstrate that the robustness of our definitions. One important future work is to prove that operators such as external choice, interleaving, or monadic composition respect equalities with respect to the various semantics, especially DRW-bisimilarity.

We plan to introduce a new type `Process`, which has as additional parameter a code for a category of processes, i.e. $+$, p , or ∞ . `Process+` would then be `Process +`, `Process ∞` would then be `Process ∞` , and our original `Process` would now be `Process p` . This was actually a suggestion by of the anonymous referees for one of our papers. First experiments show that this could work and not lead to problems with the termination checker.

What needs to be seen is whether CSP-Agda can be used for programming in Agda realistically. This could connect with the work by Abel et al. [2017, 2016] on implementing graphical user interfaces, which are concurrent in nature, in Agda. In general it seems that CSP is more used for modelling existing concurrent programs, and then determining properties of the original program, rather than writing programs directly in CSP. Frameworks exist for this such as JCSP (Welch, P. H. and Austin, P. D. [1999]) and CSP++ (Gardner [2000]), but we have not yet found good examples of actually writing programs directly in those frameworks rather than using them for modelling behaviour.

CSP-Agda-Simulator is a rather simple tool at the moment. What we plan is to combine it with an approach for representing GUIs in Agda by Abel et al. [2017, 2016], so that one can see several traces at the same time.




Bibliography

- E. Aaron. A user-level introduction to the Nuprl proof development system. Technical Report (CIS) 822, University of Pennsylvania, Department of Computer Science, 2001. URL http://repository.upenn.edu/cis/_reports/822.
- A. E. Abdallah. *Communicating Sequential Processes. The first 25 years: Symposium on the occasion of 25 years of CSP, London, UK, July 7-8, 2004. Revised invited papers*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN 978-3-540-32265-8.
- A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. URL <http://www2.tcs.ifi.lmu.de/~abel/publications.html>.
- A. Abel. Compositional coinduction with sized types. In I. Hasuo, editor, *Coalgebraic Methods in Computer Science*, pages 5–10. Springer, 2016. ISBN 978-3-319-40370-0. doi: 10.1007/978-3-319-40370-0_2. URL http://dx.doi.org/10.1007/978-3-319-40370-0_2.
- A. Abel and J. Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In P. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, Grenoble, France, 12 April 2014*, volume 153 of *Electronic Proceedings in Theoretical Computer Science*, pages 51–67. Open Publishing Association, 2014. doi: 10.4204/EPTCS.153.4.
- A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13*, pages 185–196. ACM, 2013. doi: 10.1145/2500365.2500591.

-
- A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 27–38. ACM, 2013. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429075. URL <http://doi.acm.org/10.1145/2429069.2429075>.
- A. Abel, S. Adelsberger, and A. Setzer. ooAgda. Agda Library, 2016. URL <https://github.com/agda/ooAgda>.
- A. Abel, S. Adelsberger, and A. Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming*, 27:38, Jan 2017. doi: 10.1017/S0956796816000319. URL <https://doi.org/10.1017/S0956796816000319>.
- Agda Community. The Agda Wiki, 2017a. URL <http://wiki.portal.chalmers.se/agda/pmwiki.php>.
- Agda Community. Literal Agda, 2017b. URL <http://agda.readthedocs.io/en/latest/tools/literate-programming.html>.
- R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1997. URL <http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/home/ftp/1997/CMU-CS-97-144.pdf>. CMU-CS-97-144.
- T. Altenkirch and N. A. Danielsson. Termination checking nested inductive and coinductive types. Slides of talk given at Proceedings of workshop on partiality and recursion in interactive theorem provers, PAR'10, <http://staff.computing.dundee.ac.uk/katya/par-10/thorsten.pdf>, 2010.
- R. J. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In *Stepwise refinement of distributed systems models, formalisms, correctness: REX workshop, Mook, The Netherlands May 29 – June 2, 1989. Proceedings*, pages 42–66. Springer, 1990. ISBN 978-3-540-47035-9. doi: 10.1007/3-540-52559-9_60. URL https://doi.org/10.1007/3-540-52559-9_60.
- J. C. M. Baeten, D. A. van Beek, and J. E. Rooda. Process algebra. In *Handbook of dynamic system modeling*. CRC Press, 2007. URL <http://mate.tue.nl/mate/pdfs/8509.pdf>.
-

-
- B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer, 2007. ISBN 3-540-68977-X, 978-3-540-68977-5.
- M. Benke. Alonzo – a compiler for Agda. Talk given at Agda Implementors’ Meeting 6, Gothenburg, Sweden, May 24 – 30, 2007, 2007. URL <http://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf>.
- J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109 – 137, 1984a. ISSN 0019-9958. doi: [http://dx.doi.org/10.1016/S0019-9958\(84\)80025-X](http://dx.doi.org/10.1016/S0019-9958(84)80025-X). URL <http://www.sciencedirect.com/science/article/pii/S001999588480025X>.
- J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. CWI Technical Report IW 206/82, Stichting Mathematisch Centrum, Amsterdam, 1982. URL <http://oai.cwi.nl/oai/asset/6750/6750A.pdf>. Preprint.
- J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and control*, 60(1):109–137, 1984b. doi: 10.1016/S0019-9958(84)80025-X. URL <http://www.sciencedirect.com/science/article/pii/S001999588480025X>.
- Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq’Art. The Calculus of Inductive Constructions*. Springer, 1st edition, 2010. ISBN 3642058809, 9783642058806.
- M. Bezem, R. Bol, and J. F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9(1):1–48, Jan 1997. ISSN 1433-299X. doi: 10.1007/BF01212523. URL <https://doi.org/10.1007/BF01212523>.
- R. Bird and P. Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. ISBN 0-13-484189-1.
- S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software: Third International Symposium, TACS’97 Sendai, Japan, September 23–26, 1997 Proceedings*, pages 515–529. Springer, 1997. ISBN 978-3-540-69530-1. doi: 10.1007/BFb0014565. URL <https://doi.org/10.1007/BFb0014565>.
- A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *Proceedings of the 22Nd*
-

- International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78. Springer, 2009. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_6. URL http://dx.doi.org/10.1007/978-3-642-03359-9_6.
- E. Brady. Idris, a language with dependent types – Extended abstract, 2008. URL <http://www.cs.st-and.ac.uk/~eb/drafts/ifl08.pdf>.
- E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X. URL http://journals.cambridge.org/article_S095679681300018X.
- E. C. Brady. Idris —: Systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0. doi: 10.1145/1929529.1929536. URL <http://doi.acm.org/10.1145/1929529.1929536>.
- S. D. Brookes and A. Roscoe. An improved failures model for communicating processes. In *International Conference on Concurrency*, pages 281–305. Springer, 1984. doi: 10.1007/3-540-15670-4_14. URL https://link.springer.com/chapter/10.1007%2F3-540-15670-4_14.
- S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984. ISSN 0004-5411. doi: 10.1145/828.833. URL <http://doi.acm.org/10.1145/828.833>.
- N. C. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. In *The thirty-first Communicating Process Architectures Conference, CPA 2008*, pages 67–83, 2008. doi: 10.3233/978-1-58603-907-3-67. URL <http://dx.doi.org/10.3233/978-1-58603-907-3-67>.
- N. C. C. Brown. Automatically generating CSP models for communicating Haskell processes. *Electronic Communications of the EASST*, 23, 2009. doi: 10.14279/tuj.eceasst.23.325. URL <http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/325>.
- J. Cederquist. *A pointfree approach to Constructive Analysis in Type Theory*. PhD thesis, Department of Computing Science, Cahlemers University of Technology and University of Göteborg, Sweden, 1997. URL <http://wslc.math.ist.utl.pt/ftp/pub/CederquistJ/97-C-PhDthesis.pdf>.

-
- J. Cederquist, T. Coquand, and S. Negri. The Hahn-Banach theorem in type theory. In G. Sambin and J. M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 57 – 72. Oxford University Press, 1998.
- J. M. Chapman. *Type checking and normalisation*. PhD thesis, Dept. of Computer Science, University of Nottingham, UK, 2009. URL <http://eprints.nottingham.ac.uk/10824/>.
- A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltev. Formal verification and validation of ertms industrial railway train spacing system. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 378–393. Springer, 2012. ISBN 978-3-642-31424-7. doi: 10.1007/978-3-642-31424-7_29. URL https://doi.org/10.1007/978-3-642-31424-7_29.
- E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996. ISSN 0360-0300. doi: 10.1145/242223.242257. URL <http://doi.acm.org/10.1145/242223.242257>.
- R. Cleaveland and P. Panangaden. Type theory and concurrency. *International Journal of Parallel Programming*, 17(2):153–206, Apr 1988. ISSN 1573-7640. doi: 10.1007/BF01383954. URL <https://doi.org/10.1007/BF01383954>.
- Coq Community. The Coq Proof Assistant, 2015. URL <https://coq.inria.fr/>.
- Coq Development Team. The Coq proof assistant. Reference manual, 2015. URL <https://coq.inria.fr/distrib/current/refman/>.
- C. Coquand. Agda: An interactive proof editor. Homepage for Agda version 1, 14 January 2009. URL <http://ocvs.cfv.jp/Agda/index.html>.
- T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer, 1994. doi: 10.1007/3-540-58085-9_72. URL http://dx.doi.org/10.1007/3-540-58085-9_72.
- T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, Feb. 1988. ISSN 0890-5401. doi: 10.1016/0890-5401(88)90005-3. URL [http://dx.doi.org/10.1016/0890-5401\(88\)90005-3](http://dx.doi.org/10.1016/0890-5401(88)90005-3).
-

-
- D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop, London 1992: Proceedings of the Seventh Annual Z User Meeting, London 14–15 December 1992*, pages 1–5. Springer, 1993. ISBN 978-1-4471-3556-2. doi: 10.1007/978-1-4471-3556-2_1. URL http://dx.doi.org/10.1007/978-1-4471-3556-2_1.
- L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982. ISBN 0-89791-065-6. doi: 10.1145/582153.582176. URL <http://doi.acm.org/10.1145/582153.582176>.
- N. A. Danielsson. Beating the productivity checker using embedded languages, 2010. URL <https://arxiv.org/abs/1012.4898>. arXiv:1012.4898.
- N. A. Danielsson and T. Altenkirch. Mixing induction and coinduction. Draft. <https://pdfs.semanticscholar.org/bd3f/7067d8ad16d877f4f3e1428a6e21ce771293.pdf>, 2009.
- N. A. Danielsson and T. Altenkirch. Subtyping, declaratively. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21–23, 2010. Proceedings*, pages 100–118. Springer, 2010. ISBN 978-3-642-13321-3. doi: 10.1007/978-3-642-13321-3_8. URL https://doi.org/10.1007/978-3-642-13321-3_8.
- N. A. Danielsson and U. Norell. Parsing mixfix operators. In S.-B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages: 20th International Symposium, IFL 2008, Hatfield, UK, September 10–12, 2008. Revised Selected Papers*, pages 80–99. Springer, 2011. ISBN 978-3-642-24452-0. doi: 10.1007/978-3-642-24452-0_5. URL https://doi.org/10.1007/978-3-642-24452-0_5.
- R. De Nicola. A gentle introduction to process algebras*. Notes, Last accessed 2014. URL <http://www.pst.ifi.lmu.de/Lehre/sose-2013/formale-spezifikation-und-verifikation/intro-to-pa.pdf>.
- E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
-

-
- G. Dowek, A. Felty, H. Herbelin, G. Huet, B. Werner, and C. Paulin-Mohring. The Coq proof assistant user's guide: Version 5.6, 1991. URL <https://hal.inria.fr/inria-00070034>. INRIA.
- P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical frameworks*, pages 280 – 306. Cambridge University Press, 1991.
- P. Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 workshop on types for proofs and programs, Båstad*, pages 106–114, June 1992. Available from <http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz>.
- P. Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994. ISSN 0934-5043. doi: 10.1007/BF01211308. URL <http://dx.doi.org/10.1007/BF01211308>.
- P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525 – 549, June 2000. doi: 10.2307/2586554. URL <https://doi.org/10.2307/2586554>.
- P. Dybjer and A. Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003. doi: 10.1016/S0168-0072(02)00096-9. URL <http://www.sciencedirect.com/science/article/pii/S0168007202000969>.
- S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, Jan 1998. ISSN 0098-5589. doi: 10.1109/32.663994.
- A. S. Elliott. A concurrency system for IDRIS & ERLANG. Bachelors Dissertation, University of St Andrews, 2015. URL <http://lenary.co.uk/publications/dissertation/>.
- P. A. Emrath, S. Chosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pages 580–588. ACM, 1989. ISBN 0-89791-341-8. doi: 10.1145/76263.76329. URL <http://doi.acm.org/10.1145/76263.76329>.
- ERTMS. The European Rail Traffic Mangement System, 2013. URL <http://www.ertms.net/>.
-

-
- J. Faber and R. Meyer. Model checking data-dependent real-time properties of the european train control system. In *2006 Formal Methods in Computer Aided Design*, pages 76–77, Nov 2006. doi: 10.1109/FMCAD.2006.21. URL <http://ieeexplore.ieee.org/document/4021012/>.
- R. W. Floyd. Assigning meanings to programs. In T. R. Colburn, J. H. Fetzer, and T. L. Rankin, editors, *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993. ISBN 978-94-011-1793-7. doi: 10.1007/978-94-011-1793-7_4. URL http://dx.doi.org/10.1007/978-94-011-1793-7_4.
- M. Fontaine. *A Model Checker for CSP-M*. PhD thesis, Matheematisch-Naturwissenschaftliche Fakultät, Heinrich-Heine-Universität Düsseldorf, Germany, 2011. URL http://docserv.uni-duesseldorf.de/servlets/DerivateServlet/Derivate-21671/dissertation_marc_fontaine.pdf.
- Formal Systems (Europe) Ltd. Process Behaviour Explorer. ProBE user manual, 2003. URL http://www.computing.surrey.ac.uk/personal/st/H.Treharne/teaching/com2007_2012/resources/probe-manual-2.ps.
- K. Fowler. *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes, 2009. ISBN 978-0750685672.
- W. B. Gardner. *CSP++: An object-oriented application framework for software synthesis from CSP specifications*. PhD thesis, University of Victoria, Victoria, Canada, 2000. URL <http://www.uoguelph.ca/~gardnerw/pubs/gardner.pdf>.
- S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: languages and tools for formal specification*. Springer Texts and Monographs in Computer Science. Springer, 1993. ISBN 978-1-4612-2704-5. URL <https://link.springer.com/book/10.1007%2F978-1-4612-2704-5>.
- M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development: 6th International Joint Conference CAAP/FASE Aarhus, Denmark, May 22–26, 1995 Proceedings*, pages 82–96. Springer, 1995. ISBN 978-3-540-49233-7. doi: 10.1007/3-540-59293-8_188. URL http://dx.doi.org/10.1007/3-540-59293-8_188.
-

-
- E. Gimenez. Co-inductive types in Coq: An experiment with the alternating bit protocol, 1995. URL <ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1995/RR1995-38.ps.Z>.
- S. Goncharov and L. Schröder. A coinductive calculus for asynchronous side-effecting processes. *Inf. Comput.*, 231:204–232, Oct. 2013. ISSN 0890-5401. doi: 10.1016/j.ic.2013.08.012. URL <http://dx.doi.org/10.1016/j.ic.2013.08.012>.
- S. Goncharov, L. Schröder, and C. Rauch. (Co-)algebraic foundations for effect handling and iteration. *CoRR*, abs/1405.0854, 2014. URL <http://arxiv.org/abs/1405.0854>.
- J. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology: 5th International Conference, AMAST '96 Munich, Germany, July 1–5, 1996 Proceedings*, pages 536–550. Springer, 1996. ISBN 978-3-540-68595-1. doi: 10.1007/BFb0014338. URL <https://doi.org/10.1007/BFb0014338>.
- J. F. Groote and A. Ponse. The syntax and semantics of μ CRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes: Proceedings of ACP94, the First Workshop on the Algebra of Communicating Processes, Utrecht, The Netherlands, 16–17 May 1994*, volume 94, pages 26–62. Springer, 1995. ISBN 978-1-4471-2120-6. doi: 10.1007/978-1-4471-2120-6_2. URL https://doi.org/10.1007/978-1-4471-2120-6_2.
- P. Hancock and A. Setzer. The IO monad in dependent type theory, 1999. URL <http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html>. Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27–28 March 1999.
- P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, LNCS, Vol. 1862, pages 317 – 331, 2000a. doi: 10.1007/3-540-44622-2_21. URL http://dx.doi.org/10.1007/3-540-44622-2_21.
- P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, pages 1 – 13, 2000b. URL <http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html>. Electronic proceedings.
-

- P. Hancock and A. Setzer. Interactive programs and weakly final coalgebras in dependent type theory. In *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, Oxford Logic Guides, pages 115 – 136, Oxford, UK, 2005. Clarendon Press. doi: 10.1093/acprof:oso/9780198566519.003.0007.
- D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987. doi: [https://doi.org/10.1016/0167-6423\(87\)90035-9](https://doi.org/10.1016/0167-6423(87)90035-9). URL <http://www.sciencedirect.com/science/article/pii/0167642387900359>.
- J. Harrison. Automated and interactive theorem proving. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Formal logical methods for system security and correctness. NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness, Marktoberdorf, Germany, 31 July-12 August 2007*, volume 14, pages 111–148, Amsterdam ; Oxford, 2008. IOS Press. ISBN 9781586038434. URL <http://www.iospress.nl>.
- P. Hausmann, A. Dijkstra, and W. Swierstra. The Utrecht Agda compiler. Talk given at TFP 2015. <http://www.staff.science.uu.nl/~swier004/publications/2015-tfp-draft.pdf>, 2015.
- R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. doi: 10.2307/1995158. URL http://www.jstor.org/stable/1995158?seq=1#page_scan_tab_contents.
- C. A. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, January 1983. ISSN 0001-0782. doi: 10.1145/357980.358021. URL <http://doi.acm.org/10.1145/357980.358021>.
- C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL <http://doi.acm.org/10.1145/363235.363259>.
- C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL <http://doi.acm.org/10.1145/359576.359585>.
- T. Hoare. Assertions in modern software engineering practice. In *Proceedings 26th Annual International Computer Software and Applications*, pages 459–459, Aug 2002. doi: 10.1109/CMP5AC.2002.1045044.

-
- T. Huisman and R. J. Boucherie. Running times on railway sections with heterogeneous train traffic. *Transportation Research Part B: Methodological*, 35(3):271 – 292, 2001. ISSN 0191-2615. doi: [http://dx.doi.org/10.1016/S0191-2615\(99\)00051-X](http://dx.doi.org/10.1016/S0191-2615(99)00051-X). URL <http://www.sciencedirect.com/science/article/pii/S019126159900051X>.
- B. Igried. Modelling and verification of RBC handover using CSP. 26th Nordic Workshop on Programming Theory, NWPT '14 October 29-31, 2014 - Halmstad University, Sweden, 2014. URL http://ceres.hh.se/mediawiki/index.php/NWPT_2014.
- B. Igried and A. Setzer. Programming with monadic CSP-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 28–38. ACM, 2016a. ISBN 978-1-4503-4435-7. doi: 10.1145/2976022.2976032. URL <http://doi.acm.org/10.1145/2976022.2976032>.
- B. Igried and A. Setzer. CSP-Agda, 2016b. URL <http://www.cs.swan.ac.uk/~csetzer/software/agda2/cspagda/>.
- B. Igried and A. Setzer. Defining trace semantics for csp-agda, 2016c. URL <https://arxiv.org/pdf/1612.03032.pdf>. Workshop on Coalgebra, Horn Clause Logic Programming and Types Edinburgh, UK, 28–29 November 2016.
- B. Igried and A. Setzer. Representing the process algebra csp in type theory, 2016d. URL <http://www.types2016.uns.ac.rs/images/abstracts/setzer1.pdf>. 22nd International Conference on Types for Proofs and Programs, TYPES 2016, Novi Sad, Serbia, 23-26 May 2016.
- B. Igried and A. Setzer. Strong bisimilarity implies trace semantics in csp-agda., 2016e. URL <http://types2017.elte.hu/proc.pdf>. 23rd International Conference on Types for Proofs and Programs, TYPES 2017, Budapest, Hungary, 29 May - 1 June 2017.
- B. Igried and A. Setzer. Defining trace semantics for CSP-Agda, 30 Jan 2018. URL <http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf>. Accepted for publication in Postproceedings TYPES 2016, 23 pp, available from <http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf>.
-

-
- Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–123. Springer, 2005. doi: 10.1007/978-3-540-31980-1_8. URL https://link.springer.com/chapter/10.1007/978-3-540-31980-1_8.
- C. B. Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990. ISBN 978-0138807337.
- S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. ISBN 978-0521826143.
- F. Kammüller. CSP revisited, 2007. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.171.657&rep=rep1&type=pdf>. Theorem Proving in Higher Order Logics.
- K. Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea, UK, August 2012. URL <http://www.swan.ac.uk/csetzer/articlesFromOthers/index.html>.
- K. Kanso and A. Setzer. A light-weight integration of automated and interactive theorem proving. *Mathematical Structures in Computer Science*, FirstView:1–25, 12 November 2014. ISSN 1469-8072. doi: 10.1017/S0960129514000140. URL http://journals.cambridge.org/article_S0960129514000140.
- J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, May 2002. ISBN 1-58113-472-X.
- L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177726. URL <http://doi.acm.org/10.1145/177492.177726>.
- A. v. Lamsweerde. From system goals to software architecture. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003. Advanced Lectures*, pages 25–43. Springer, 2003. ISBN 978-3-540-39800-4. doi: 10.1007/978-3-540-39800-4_2. URL https://doi.org/10.1007/978-3-540-39800-4_2.
-

-
- K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 382–398, 2014. doi: 10.1007/978-3-319-06410-9_27. URL http://dx.doi.org/10.1007/978-3-319-06410-9_27.
- X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7): 107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL <http://doi.acm.org/10.1145/1538788.1538814>.
- N. López, M. Núñez, and F. Rubio. Stochastic process algebras meet Eden. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods: Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings*, pages 29–48. Springer, 2002. ISBN 3-540-43703-7. doi: 10.1007/3-540-47884-1_3. URL https://doi.org/10.1007/3-540-47884-1_3.
- N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151. ACM, 1987. ISBN 0-89791-239-X. doi: 10.1145/41840.41852. URL <http://doi.acm.org/10.1145/41840.41852>.
- L. Magnusson and B. Nordström. The Alf proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93 Nijmegen, The Netherlands, May 24–28, 1993 Selected Papers*, pages 213–237. Springer, 1994. ISBN 978-3-540-48440-0. doi: 10.1007/3-540-58085-9_78. URL https://doi.org/10.1007/3-540-58085-9_78.
- Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 2012.
- P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages Supplement C, 73 – 118. Elsevier, 1975. doi: [https://doi.org/10.1016/S0049-237X\(08\)71945-1](https://doi.org/10.1016/S0049-237X(08)71945-1). URL <http://www.sciencedirect.com/science/article/pii/S0049237X08719451>.
- P. Martin-Löf. Constructive mathematics and computer programming. In H. P. L. Jonathan Cohen, Jerzy Łoś and K.-P. Podewski, editors, *Logic*,
-

- Methodology and Philosophy of Science VI* Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153 – 175. Elsevier, 1982. doi: [http://dx.doi.org/10.1016/S0049-237X\(09\)70189-2](http://dx.doi.org/10.1016/S0049-237X(09)70189-2). URL <http://www.sciencedirect.com/science/article/pii/S0049237X09701892>.
- P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984. ISBN 88-7088-105-9.
- C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi: 10.1017/S0956796803004829.
- A. I. S. McInnes. *A formal approach to specifying and verifying spacecraft behavior*. PhD thesis, Electrical and Computer Engineering, Utah State University, 2007. URL <https://search.proquest.com/docview/304805889?accountid=14680>.
- R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. ISSN 0022-0000. doi: [http://dx.doi.org/10.1016/0022-0000\(78\)90014-4](http://dx.doi.org/10.1016/0022-0000(78)90014-4). URL <http://www.sciencedirect.com/science/article/pii/0022000078900144>.
- R. Milner. *A Calculus of Communicating Systems*. Springer, 1982. ISBN 0387102353.
- R. Milner. *The definition of standard ML: revised*. MIT press, 1997. ISBN 978-0262132558.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992. ISSN 0890-5401. doi: [http://dx.doi.org/10.1016/0890-5401\(92\)90008-4](http://dx.doi.org/10.1016/0890-5401(92)90008-4). URL <http://www.sciencedirect.com/science/article/pii/0890540192900084>.
- L. Mo. *Refinement Methods for Software Architecture Design Using the Software Architecture Model*. PhD thesis, Computer Science, Miami, FL, USA, 2005. AAI3169465, <https://search.proquest.com/docview/305355896>.
- E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. ISSN 0890-5401. doi: [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4). URL <http://www.sciencedirect.com/science/article/pii/0890540191900524>.

-
- C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, July 1988. ISSN 0164-0925. doi: 10.1145/44501.44503. URL <http://doi.acm.org/10.1145/44501.44503>.
- J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer programming*, 9(3):287–306, 1987. doi: [https://doi.org/10.1016/0167-6423\(87\)90011-6](https://doi.org/10.1016/0167-6423(87)90011-6). URL <http://www.sciencedirect.com/science/article/pii/0167642387900116>.
- T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006. doi: 10.1016/j.jlap.2005.09.006. URL <http://dx.doi.org/10.1016/j.jlap.2005.09.006>.
- R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992. ISSN 1057-4514. doi: 10.1145/130616.130623. URL <http://doi.acm.org/10.1145/130616.130623>.
- G. Newman. Railway signalling system, Aug. 1 1995. URL <https://www.google.com/patents/US5437422>. US Patent 5,437,422.
- T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002. URL 978-3540433767.
- B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf’s Type Theory: An Introduction*. Clarendon Press, 1990. ISBN 0-19-853814-6.
- U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007. URL <http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf>.
- U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures, AFP’08*, pages 230–266. Springer, 2009a. ISBN 3-642-04651-7, 978-3-642-04651-3. doi: 10.1007/978-3-642-04652-0_5. URL https://doi.org/10.1007/978-3-642-04652-0_5.
- U. Norell. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI ’09*, pages 1–2. ACM, 2009b. ISBN 978-1-60558-420-1. doi:
-

- 10.1145/1481861.1481862. URL <http://doi.acm.org/10.1145/1481861.1481862>.
- I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the first Israeli conference on computer systems engineering*, pages 4–1, 1986. URL <http://people.csail.mit.edu/rudolph/Autobiography/LRTools1986.pdf>.
- D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer, 1981. ISBN 3-540-10576-X. URL <http://dl.acm.org/citation.cfm?id=647210.720030>.
- C. Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification: LASER*, pages 45–95. Springer, 2012. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_3. URL http://dx.doi.org/10.1007/978-3-642-35746-6_3.
- L. C. Paulson. Natural deduction as higher-order resolution. *The Journal of Logic Programming*, 3(3):237–258, 1986. doi: 10.1016/0743-1066(86)90015-4. URL <http://www.sciencedirect.com/science/article/pii/0743106686900154>.
- L. C. Paulson. A preliminary users manual for Isabelle. Technical Report UCAM-CL-TR-133, University of Cambridge, Computer Laboratory, May 1988a. URL <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-133.pdf>.
- L. C. Paulson. A preliminary users manual for Isabelle. Technical Report UCAM-CL-TR-133, University of Cambridge, Computer Laboratory, May 1988b. URL <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-133.pdf>.
- L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994. ISBN 978-3540582441.
- L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, pages 1–10, 2010. URL <https://www21.in.tum.de/~blanchet/iwil2010-sledgehammer.pdf>.

-
- K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *CTCS'89*, pages 128–140, London, UK, 1989. Springer-Verlag.
- S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993. URL <https://www.microsoft.com/en-us/research/wp-content/uploads/1993/03/grasp-jfit.pdf>.
- S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84. ACM, 1993. ISBN 0-89791-560-7. doi: 10.1145/158511.158524.
- A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57. IEEE, Oct 1977. doi: 10.1109/SFCS.1977.32.
- K. V. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2-3):285–327, 1995. ISSN 0167-6423. doi: [https://doi.org/10.1016/0167-6423\(95\)00017-8](https://doi.org/10.1016/0167-6423(95)00017-8). URL <http://www.sciencedirect.com/science/article/pii/0167642395000178>. Selected Papers of ESOP'94, the 5th European Symposium on Programming.
- A. W. Roscoe. *The theory and practice of concurrency*, volume 169. Prentice Hall Englewood Cliffs, 1998. ISBN 978-0136744092.
- A. W. Roscoe. *Understanding concurrent systems*. Springer, 2010. ISBN 978-1848822573.
- J. Rushby. Formal methods and critical systems in the real world. In D. Craigen and K. Summerskill, editors, *Formal Methods for Trustworthy Computer Systems (FM89): Report from FM89: A Workshop on the Assessment of Formal Methods for Trustworthy Computer Systems. 23–27 July 1989, Halifax, Canada*, pages 121–125. Springer, 1989. ISBN 978-1-4471-3532-6. URL <https://pdfs.semanticscholar.org/6312/c15242f7748013f2ef5044479e112dd8a26e.pdf>. Appendix C.1.
- J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, Oct. 2000. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00056-6. URL [http://dx.doi.org/10.1016/S0304-3975\(00\)00056-6](http://dx.doi.org/10.1016/S0304-3975(00)00056-6).
-

- P. Ryan and S. A. Schneider. *The modelling and analysis of security protocols: the CSP approach*. Addison-Wesley Professional, 2001. ISBN 9780201674712.
- S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley, 1st edition, 1999. ISBN 0471623733.
- M. P. A. Sellink. Verifying process algebra proofs in type theory. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Semantics of Specification Languages (SoSL): Proceedings of the International Workshop on Semantics of Specification Languages, Utrecht, The Netherlands, 25 – 27 October 1993*, pages 315–339. Springer, 1994. ISBN 3-540-19854-7. doi: 10.1007/978-1-4471-3229-5_18. URL https://doi.org/10.1007/978-1-4471-3229-5_18.
- A. Setzer. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*, pages 1 – 16, 2006. URL <http://www.cs.nott.ac.uk/~nhn/TFP2006/TFP2006-Programme.html>.
- A. Setzer. How to reason coinductively informally. In R. Kahle, T. Strahm, and T. Studer, editors, *Advances in Proof Theory*, pages 377–408. Springer, 2016. ISBN 978-3-319-29198-7. doi: 10.1007/978-3-319-29198-7_12. URL http://dx.doi.org/10.1007/978-3-319-29198-7_12.
- A. Setzer and P. Hancock. Interactive programs and weakly final coalgebras in dependent type theory (extended version). In *Dependently Typed Programming 2004*, pages 1 – 30, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany. URL <http://drops.dagstuhl.de/opus/volltexte/2005/176/>.
- A. Setzer and B. Igried. Trace and stable failures semantics for CSP-Agda. Accepted for publication in Post proceedings on Workshop on Coalgebra, Horn Clause Logic Programming and Types, Electronic Proceedings in Theoretical Computer Science, available at <http://www.cs.swan.ac.uk/~csetzer/articles/coalp2017SetzerIgriedTraceStableFailuresSemantics.pdf>, May 2017. URL <http://www.cs.swan.ac.uk/~csetzer/articles/coalp2017SetzerIgriedTraceStableFailuresSemantics.pdf>.
- A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi*, volume 8560 of *LNCS*, pages 31–45. Springer, 2014. ISBN 978-3-319-08917-1.



doi: 10.1007/978-3-319-08918-8_3. URL http://dx.doi.org/10.1007/978-3-319-08918-8_3.

M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000 Austin, TX, USA, November 1–3, 2000 Proceedings*, FMCAD '00, pages 108–125. Springer, 2000. ISBN 3-540-41219-0. doi: 10.1007/3-540-40922-X_8. URL https://doi.org/10.1007/3-540-40922-X_8.

R. Spadotti. A mechanized theory of regular trees in dependent type theory. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24–27, 2015, Proceedings*, pages 405–420. Springer, 2015. ISBN 978-3-319-22102-1. doi: 10.1007/978-3-319-22102-1_27. URL https://doi.org/10.1007/978-3-319-22102-1_27.

J. Spivey. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988. ISBN 9780521054140.

N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1996. ISBN 0201427877.

A. Stump. *Verified Functional Programming in Agda*. Morgan & Claypool, 2016. ISBN 978-1970001242. URL <https://svn.divms.uiowa.edu/repos/clc/projects/agda/book/book.pdf>. Code available from <http://homepage.divms.uiowa.edu/~astump/>.

H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods: 4th International Symposium of Formal Methods Europe Graz, Austria, September 15–19, 1997 Proceedings*, pages 318–337. Springer, 1997. ISBN 978-3-540-69593-6. doi: 10.1007/3-540-63533-5_17. URL https://doi.org/10.1007/3-540-63533-5_17.

Thomas Hallgren. Alfa, 2017. URL <http://www.cse.chalmers.se/~hallgren/Alfa/>.

D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, July 2004. doi: 10.3217/jucs-010-07-0751. URL http://www.jucs.org/jucs_10_7/total_functional_programming.



-
- University of Oxford. FDR2 user's manual version 2.82, 2012. URL <https://www.cs.ox.ac.uk/projects/concurrency-tools/fdr-2.94-html-manual/fdr2manual.html>.
- P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91592. URL <http://doi.acm.org/10.1145/91556.91592>.
- P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text*, pages 24–52, Berlin, Heidelberg, 1995. Springer. ISBN 978-3-540-49270-2. doi: 10.1007/3-540-59451-5_2. URL https://doi.org/10.1007/3-540-59451-5_2.
- P. Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, September 1997. ISSN 0360-0300. doi: 10.1145/262009.262011. URL <http://doi.acm.org/10.1145/262009.262011>.
- P. Wadler. The marriage of effects and monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 63–74. ACM, 1998. ISBN 1-58113-024-4. doi: 10.1145/289423.289429. URL <http://doi.acm.org/10.1145/289423.289429>.
- M. P. Ward and K. H. Bennett. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering*, 5(01):25–47, 1995. doi: 10.1142/S0218194095000034. URL <http://www.worldscientific.com/doi/abs/10.1142/S0218194095000034>.
- P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. Sputh. Integrating and extending JCSP. In S. Schneider, A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, July 2007. URL <http://kar.kent.ac.uk/24001/>.
- Welch, P. H. and Austin, P. D. The JCSP (CSP for Java) Home Page, 1999. URL <https://www.cs.kent.ac.uk/projects/ofa/jcsp/>.
- K. Winter. Model checking railway interlocking systems. *Aust. Comput. Sci. Commun.*, 24(1):303–310, Jan. 2002. doi: 10.1145/563857.563836. URL <http://dx.doi.org/10.1145/563857.563836>.
-



J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, Oct. 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592436. URL <http://doi.acm.org/10.1145/1592434.1592436>.

E. J. Younger, Z. Luo, K. H. Bennett, and T. M. Bull. Reverse engineering concurrent programs using formal modelling and analysis. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 239–248, Nov 1996. doi: 10.1109/WCRE.1996.558918.







Appendices



Appendix A

Agda Code

A.1 addTick.agda

```
--@PREFIX@addtick
```

```
module addTick where
```

```
open import Size
open import Data.String renaming (_++_ to _++s_)
open import Data.Fin
open import Data.Sum
open import process
open import choiceSetU
open import showFunction
open import dataAuxFunction
open import labelUniv
open import showLabelP
open import renamingResult
open import internalChoice
```

```
--@BEGIN@addtickDef
```

```
2-✓Str : {c0 c1 : Choice} → (a : ChoiceSet c0)
      → (b : ChoiceSet c1) → String
2-✓Str a b = "(2-✓ " ++s choice2Str a ++s " " ++s choice2Str b ++s ")"
```

```
2-✓+ : ∀ {i} → {c0 c1 : Choice} → (a : ChoiceSet c0)
      → {lu : LUniv}
      → (b : ChoiceSet c1) → Process+ i {lu} (c0 ⊞' c1)
```

```

E    (2-✓ + a b)      = ∅'
Lab  (2-✓ + a b)      ()
PE   (2-✓ + a b)      ()
I    (2-✓ + a b)      = ∅'
PI   (2-✓ + a b)      ()
T    (2-✓ + a b)      = fin 2
PT   (2-✓ + a b)      zero = inj₁ a
PT   (2-✓ + a b)      (suc zero) = inj₂ b
PT   (2-✓ + a b)      (suc (suc ()))
Str+ (2-✓ + a b)      = "(2-✓ " ++s choice2Str a ++s "
                                " ++s choice2Str b ++s ")"

```

```

2-✓ : ∀ {i} → {c₀ c₁ : Choice} → (a : ChoiceSet c₀) → {lu : LUniv}
      → (b : ChoiceSet c₁) → Process i {lu} (c₀ ⊕' c₁)

```

```

2-✓ a b = node (2-✓ + a b)

```

```

2-✓∞ : ∀ {i} → {c₀ c₁ : Choice} → (a : ChoiceSet c₀) → {lu : LUniv}
      → (b : ChoiceSet c₁) → Process∞ i {lu} (c₀ ⊕' c₁)
forcep (2-✓∞ a b) = (2-✓ a b)
Str∞ (2-✓∞ a b)   = 2-✓Str a b

```

```

--@END

```

```

unifyA⊕A : {A : Set} → A ⊕ A → A
unifyA⊕A (inj₁ a) = a
unifyA⊕A (inj₂ a) = a

```

```

add✓Str : ∀ {c : Choice} → (a : ChoiceSet c)
      → String → String
add✓Str a str = "(add✓ " ++s choice2Str a ++s "
                                " ++s str ++s ")"

```

```

--@BEGIN@addTickPartOneDef

```

```

mutual

```

```
add✓∞ : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
      → Process∞ i {lu} c → Process∞ i {lu} c
```

```
forcep (add✓∞ a P) = add✓ a (forcep P)
Str∞ (add✓∞ a P) = add✓Str a (Str∞ P)
```

```
add✓ : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
      → Process i {lu} c → Process i {lu} c
add✓ a (terminate b) = fmap unifyA⊕A (2-✓ a b)
add✓ a (node P)      = node (add✓+ a P)
```

```
add✓+ : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
      → Process+ i {lu} c → Process+ i {lu} c
```

```
E (add✓+ a P)      = E P
Lab (add✓+ a P)     = Lab P
PE (add✓+ a P) s    = add✓∞ a (PE P s)
I (add✓+ a P)       = I P
PI (add✓+ a P) s    = add✓∞ a (PI P s)
T (add✓+ a P)       = T' ⊕' T P
PT (add✓+ a P) (inj1 _) = a
PT (add✓+ a P) (inj2 c) = PT P c
Str+ (add✓+ a P)    = add✓Str a (Str+ P)
```

```
--@END
```

```
--@BEGIN@addTimedDef
```

```
addTimed✓Str : {c : Choice} → (a : ChoiceSet c)
              → String → String
addTimed✓Str a str = "(addTimed✓ " ++s choice2Str a ++s "
                      " ++s str ++s ")"
```

```
mutual
```

```
addTimed✓∞ : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
      → Process∞ i {lu} c → Process∞ i {lu} c
forcep (addTimed✓∞ a P) = addTimed✓ a (forcep P)
Str∞ (addTimed✓∞ a P) = addTimed✓Str a (Str∞ P)
```

```

addTimed✓ : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
           → Process i {lu} c → Process i {lu} c
addTimed✓ a (terminate b) = fmap unifyA⊕A (2-✓ a b)
addTimed✓ a (node P) = node (addTimed✓ + a P)

```

```

addTimed✓ + : ∀ {i} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv}
           → Process+ i {lu} c
           → Process+ i {lu} c
E   (addTimed✓ + a P) = E P
Lab (addTimed✓ + a P) = Lab P
PE  (addTimed✓ + a P) s = PE P s
I   (addTimed✓ + a P) = I P
PI  (addTimed✓ + a P) s = addTimed✓ ∞ a (PI P s)
T   (addTimed✓ + a P) = T' ⊕' T P
PT  (addTimed✓ + a P) (inj1 _) = a
PT  (addTimed✓ + a P) (inj2 c) = PT P c
Str+ (addTimed✓ + a P) = addTimed✓ Str a (Str+ P)

```

```
--@END
```

A.2 addTickOperator.agda

```
module addTickOperator where
```

```

open import Size
open import process
open import choiceSetU
open import primitiveProcess
open import Data.Sum
open import Data.String renaming (_++_ to _++s_)
open import labelUniv

```

```

choiceFunctionAddEl : {ca : Choice} → (c : Choice)
                   → (f : ChoiceSet c → ChoiceSet ca)
                   → ChoiceSet ca

```

```

      → ChoiceSet (fin 1 ⊕' c)
      → ChoiceSet ca
choiceFunctionAddEl c f a (inj1 x) = a
choiceFunctionAddEl c f a (inj2 y) = f y

```

mutual

```

addTickStr : {ca : Choice} → (c : Choice)
           → (f : ChoiceSet c → ChoiceSet ca)
           → String → String
addTickStr c f s = "Something" ++ s s

addTick∞ : {ca : Choice} → {i : Size} → (c : Choice)
          → {lu : LUniv}
          → (f : ChoiceSet c → ChoiceSet ca)
          → Process∞ i {lu} ca
          → Process∞ i {lu} ca
forcep (addTick∞ c f P) {j} = addTick c f (forcep P {j})
Str∞ (addTick∞ c f P) = addTickStr c f (Str∞ P)

addTick : {ca : Choice} → {i : Size} → (c : Choice)
        → {lu : LUniv}
        → (f : ChoiceSet c → ChoiceSet ca)
        → Process i {lu} ca
        → Process i {lu} ca
addTick {ca} {i} c f (terminate a) = MSKIP i ca (fin 1 ⊕' c)
                                   (choiceFunctionAddEl c f a)
addTick c f (node Q) = node (addTick+ c f Q)

addTick+ : {ca : Choice} → {i : Size} → (c : Choice)
          → {lu : LUniv}
          → (f : ChoiceSet c → ChoiceSet ca)
          → Process+ i {lu} ca
          → Process+ i {lu} ca
E (addTick+ c f P) = E P
Lab (addTick+ c f P) x = Lab P x
PE (addTick+ c f P) x = PE P x
I (addTick+ c f P) = I P
PI (addTick+ c f P) x = PI P x
T (addTick+ c f P) = c ⊕' T P
PT (addTick+ c f P) (inj1 x) = f x

```

```

PT (addTick+ c f P) (inj2 y) = PT P y
Str+ (addTick+ c f P) = addTickStr c f (Str+ P)

```

A.3 auxData.agda

```

--@PREFIX@auxdata
module auxData where

open import Data.Bool
open import Data.String renaming (_==_ to _==strb_)
open import Data.Product hiding (_×_ ; Σ)
open import Level

--@BEGIN@prod

data _×_ (a b : Set) : Set where
  _,_ : a → b → a × b

--@END
infixr 5 _×_

proj1' : {A B : Set} (ab : A × B) → A
proj1' (a , b) = a

proj2' : {A B : Set} (ab : A × B) → B
proj2' (a , b) = b

data _∨s_ (a b : Set) : Set where
  inl : a → a ∨s b
  inr : b → a ∨s b

--@BEGIN@subset

data subset (A : Set) (f : A → Bool) : Set where
  sub : (a : A) → T (f a) → subset A f

--@END

```

```

data _==_ {A : Set} (a : A) : A → Set where
  refl : a == a

--@BEGIN@seg

record  $\Sigma$  {a b} (A : Set a) (B : A → Set b) : Set (a  $\sqcup$  b) where
  constructor _,_
  field
    proj1 : A
    proj2 : B proj1

--@END

```

A.4 auxLemmaPar.agda

```

--@PREFIX@auxlemmapar

module auxLemmaPar where

-- uses normal label not labelUniv

open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import lemFmap
open import auxData
open import dataAuxFunction
open import Data.Product
open import labelUniv
open import parallelSimple
open import restrict
open import Data.Bool

```

```

open import labelEq
open import Agda.Builtin.Equality
open import Agda.Builtin.Unit
open import Data.Bool.Base renaming (T to T')
open import Data.Unit.Base
open import Data.Product

--@BEGIN@lemmaBool

lemmaBool : (a b : Bool) → T' ( a ∧ b ) → T' a
lemmaBool false b ()
lemmaBool true false ()
lemmaBool true true tt = tt

lemmaBoolR : (a b : Bool) → T' ( a ∧ b ) → T' b
lemmaBoolR false false ()
lemmaBoolR true false ()
lemmaBoolR false true ()
lemmaBoolR true true tt = tt

--@END

--@BEGIN@lemmaBoolaux

lemmaBool'aux : (a b c : Bool)
  → T' ( a ∧ b ∧ c ) → T' c
lemmaBool'aux a b c p = let
  q : T' ( b ∧ c )
  q = lemmaBoolR a ( b ∧ c ) p
  in lemmaBoolR b c q

--@END

lemmaBool' : (a b c : Bool) → T' ( a ∧ b ∧ c ) → T' c
lemmaBool' false false false ()
lemmaBool' false false true ()
lemmaBool' false true false ()
lemmaBool' false true true ()
lemmaBool' true false false ()
lemmaBool' true false true ()

```

```

lemmaBool' true true false ()
lemmaBool' true true true tt = tt

lemmaBool'' : (a b c : Bool) → T' a → T' b → T' c → T' (a ∧ b ∧ c)
lemmaBool'' false b c () x1 x2
lemmaBool'' true false c x () x2
lemmaBool'' true true c x tt x2 = x2

```

A.5 bisimForNextProcess.agda

```
--@PREFIX@bisimForNextProcess
```

```

module bisimForNextProcess where

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Empty
open import Data.Unit
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import traceImpliesTraceP
open import Data.Bool    hiding (T)
open import fdi
open import auxData

```

```
--@BEGIN@BisimForNextP
```

```

BisimForNextP : {lu : LUniv}{c : Choice} (tick tick' : Process ∞ {lu} c ⊔ ChoiceSet c)
BisimForNextP {lu}{c} (inj1 P) (inj1 P') = Bisimw {∞} P P'
BisimForNextP {lu}{c} (inj1 P) (inj2 x)      = TerminateEquivalent x P
BisimForNextP {lu}{c} (inj2 x) (inj1 P)      = TerminateEquivalent x P
BisimForNextP {lu}{c} (inj2 x) (inj2 x') = x ≡ x'

```

```
--@END
```

mutual

```

lemmaTraceTerminationEquivalentEmpty+' : {lu : LUniv}(c : Choice) (l : List (Label lu))
(P' : Process+ ∞ c)(Q : Process ∞ {lu} c ⊔
(a : ChoiceSet c)
(nodexTerEquiv : TerminateEquivalent a (node
(tr : TrP+ l Q P')
→ l ≡ []
lemmaTraceTerminationEquivalentEmpty+' c .[] P' .(inj1 (node P')) a nodexTerEquiv empty
lemmaTraceTerminationEquivalentEmpty+' c .(Lab P' x :: l) P' Q a (termeqnode terequivP)
= ⊥-elim (noExtChoice terequivP x)
lemmaTraceTerminationEquivalentEmpty+' c l P' Q a nodexTerEquiv (intc .l .Q x x1)
= lemmaTraceTerminationEquivalentEmpty+' c l P' Q a nodexTerEquiv (tnode (intc l Q
lemmaTraceTerminationEquivalentEmpty+' c .[] P' .(inj2 (PT P' x)) a nodexTerEquiv (terc x
= refl

lemmaTraceTerminationEquivalentEmpty' : {lu : LUniv}(c : Choice) (l : List (Label lu))
(P' : Process+ ∞ c)(Q : Process ∞ {lu} c ⊔
(a : ChoiceSet c)
(nodexTerEquiv : TerminateEquivalent a (node
(tr : TrP l Q (node P'))
→ l ≡ []
lemmaTraceTerminationEquivalentEmpty' c .[] P' .(inj1 (node P')) a nodexTerEquiv
(tnode empty) = refl
lemmaTraceTerminationEquivalentEmpty' c .(Lab P' x :: l) P' Q a (termeqnode terequivP) (t
= ⊥-elim (noExtChoice terequivP x)
lemmaTraceTerminationEquivalentEmpty' c l P' Q a (termeqnode terequivP) (tnode (intc .l .
= lemmaTraceTerminationEquivalentEmpty∞' c l (PI P' x) Q a (onlyIntChoice terequivP x
lemmaTraceTerminationEquivalentEmpty' c .[] P' .(inj2 (PT P' x)) a nodexTerEquiv (tnode (

lemmaTraceTerminationEquivalentEmpty∞' : {lu : LUniv}(c : Choice) (l : List (Label lu))
(P' : Process∞ ∞ c)(Q : Process ∞ {lu} c ⊔

```

```

      (a : ChoiceSet c)
      (nodexTerEquiv : TerminateEquivalent a (forcep P'))
      (tr : TrP $\infty$  l Q P')
      → l  $\equiv$  []
lemmaTraceTerminationEquivalentEmpty' $\infty$ ' c l P' Q a nodexTerEquiv tr
  = lemmaTraceTerminationEquivalentEmpty'' c l (forcep P') Q a nodexTerEquiv tr

lemmaTraceTerminationEquivalentEmpty'' : {lu : LUniv}{c : Choice} (l : List (Label lu))
  (P' : Process  $\infty$  {lu} c)(Q : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (a : ChoiceSet c)
  (nodexTerEquiv : TerminateEquivalent a P')
  (tr : TrP l Q P')
  → l  $\equiv$  []
lemmaTraceTerminationEquivalentEmpty'' c [].[(terminate x) .(inj2 x) a nodexTerEquiv (ter x) = refl]
lemmaTraceTerminationEquivalentEmpty'' c [].[(terminate x) .(inj1 (terminate x)) a nodexTerEquiv (e
lemmaTraceTerminationEquivalentEmpty'' c l .(node P') Q a nodexTerEquiv (tnode {l} {.Q} {P'} x)
  = lemmaTraceTerminationEquivalentEmpty' c l P' Q a nodexTerEquiv (tnode x)

```

mutual

```

termequivPToTicknode2 : {lu : LUniv}{c : Choice}{y : ChoiceSet c}(P : Process  $\infty$  {lu} c)
  (l : List (Label lu)) (m : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (terequivP : TerminateEquivalent y P) (tr : TrP l m P)
  → TrP l (inj1 (terminate y)) (terminate y)
termequivPToTicknode2 y (terminate .y) [].(inj1 (terminate .y)) termeqterm (empty .y) = empty y
termequivPToTicknode2 y (terminate .y) l (inj1 (node x)) termeqterm ()
termequivPToTicknode2 y (terminate .y) [].(inj2 .y) termeqterm (ter .y) = empty y
termequivPToTicknode2 y (node x) l m terequivP (tnode tr) = termequivPToTicknode2+ y x l m terequivP

termequivPToTicknode2+ : {lu : LUniv}{c : Choice}{y : ChoiceSet c}(P : Process+  $\infty$  c)
  (l : List (Label lu)) (m : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (terequivP : TerminateEquivalent y (node P)) (tr : TrP+ l m P)
  → TrP l (inj1 (terminate y)) (terminate y)
termequivPToTicknode2+ y P [].(inj1 (node P)) (termeqnode terequivP) empty = empty y
termequivPToTicknode2+ y P .(Lab P x :: l) m (termeqnode terequivP) (extc l .m x x1)
  =  $\perp$ -elim (noExtChoice terequivP x)

```

$\text{termequivPToTicknode}_2 + y \ P \ l \ m \ (\text{termeqnode } terequivP) \ (\text{intc } l \ .m \ x \ x_1)$
 $= \text{termequivPToTicknode}_2 \infty \ y \ (\text{PI } P \ x) \ l \ m \ (\text{onlyIntChoice } terequivP \ x) \ x_1$
 $\text{termequivPToTicknode}_2 + y \ P \ .[] \ .(\text{inj}_2 \ (\text{PT } P \ x)) \ (\text{termeqnode } terequivP) \ (\text{terc } x) = \text{empty } y$

$\text{termequivPToTicknode}_2 \infty : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (y : \text{ChoiceSet } c) (P : \text{Process} \infty \infty c)$
 $(l : \text{List } (\text{Label } lu)) (m : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $(terequivP : \text{TerminateEquivalent} \infty \ y \ P) (tr : \text{TrP} \infty \ l \ m \ P)$
 $\rightarrow \text{TrP } l \ (\text{inj}_1 \ (\text{terminate } y)) \ (\text{terminate } y)$

$\text{termequivPToTicknode}_2 \infty \ y \ P \ l \ m \ terequivP \ tr = \text{termequivPToTicknode}_2 \ y \ (\text{forcep } P) \ l \ m \ t$

$\text{termequivPToTick} \infty_1 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (x : \text{ChoiceSet } c) (P : \text{Process} \infty \infty c) (\text{termequivP} : \text{TerminateEquivalent} \infty \ y \ P)$
 $\rightarrow \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$

$\text{termequivPToTick} \infty_1 \{lu\} \{c\} \ x \ P \ \text{termequivP} = \text{termequivPToTick}_1 \{lu\} \{c\} \ x \ (\text{forcep } P) \ \text{termequivP}$

$\text{termequivPToTick}_1 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (x : \text{ChoiceSet } c) (P : \text{Process } \infty \{lu\} c) (\text{termequivP} : \text{TerminateEquivalent} \infty \ y \ P)$
 $\rightarrow \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$

$\text{termequivPToTick}_1 \{lu\} \{c\} \ x \ .(\text{terminate } x) \ \text{termeqterm} = \text{inj}_2 \ x$

$\text{termequivPToTick}_1 \{lu\} \{c\} \ x \ .(\text{node } Q) \ (\text{termeqnode } \{Q\} \ \text{termequivQ})$

$= \text{termequivPToTick}_1 \text{aux } \{lu\} \{c\} \ x \ Q \ \text{termequivQ} \ (\text{hasTauOrTick } \text{termequivQ})$

$\text{termequivPToTick}_1 \text{aux} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (x : \text{ChoiceSet } c) (Q : \text{Process} + \infty c)$
 $(\text{termequivQ} : \text{TerminateEquivalent} + \ x \ Q)$
 $(\text{hastauortick} : \text{ChoiceSet } (\text{I } Q) \uplus \text{ChoiceSet } (\text{T } Q))$

$\rightarrow \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$

$\text{termequivPToTick}_1 \text{aux } \{lu\} \{c\} \ x \ Q \ \text{termequivQ} \ (\text{inj}_1 \ ip)$

$= \text{termequivPToTick} \infty_1 \{lu\} \{c\} \ x \ (\text{PI } Q \ ip) \ (\text{onlyIntChoice } \text{termequivQ} \ ip)$

$\text{termequivPToTick}_1 \text{aux } \{lu\} \{c\} \ x \ Q \ \text{termequivQ} \ (\text{inj}_2 \ t) = \text{inj}_2 \ (\text{PT } Q \ t)$

$\text{termequivPToTick} \infty_2 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (x : \text{ChoiceSet } c) (P : \text{Process} \infty \infty c)$
 $(\text{termequivP} : \text{TerminateEquivalent} \infty \ x \ P)$
 $\rightarrow \text{TrP} \infty \ [] \ (\text{termequivPToTick} \infty_1 \{lu\} \{c\} \ x \ P \ \text{termequivP}) \ P$

$\text{termequivPToTick} \infty_2 \{lu\} \{c\} \ x \ P \ \text{termequivP} = \text{termequivPToTick}_2 \{lu\} \{c\} \ x \ (\text{forcep } P) \ \text{termequivP}$

$\text{termequivPToTick}_2 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (x : \text{ChoiceSet } c) (P : \text{Process } \infty \{lu\} c)$
 $(\text{termequivP} : \text{TerminateEquivalent} \ x \ P)$

```

      → TrP [] (termequivPToTick1 {lu}{c} x P termequivP) P
termequivPToTick2 {lu}{c} x .(terminate x) termeqterm = ter x
termequivPToTick2 {lu}{c} x .(node Q) (termeqnode {Q} termequivQ)
  = termequivPToTick2aux {lu}{c} x Q termequivQ (hasTauOrTick termequivQ)

termequivPToTick2aux : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(Q : Process+ ∞ c)
  (termequivQ : TerminateEquivalent+ x Q)
  (hastauortick : ChoiceSet (I Q) ⊔ ChoiceSet (T Q) )
  → TrP [] (termequivPToTick1aux {lu}{c} x Q termequivQ hastauortick)
termequivPToTick2aux {lu}{c} x Q termequivQ (inj1 ip)
  = tnode (intc [] (termequivPToTick1aux x Q termequivQ (inj1 ip)) ip)
    (termequivPToTick∞2 {lu}{c} x (PI Q ip) (onlyIntChoice termequivQ ip)))

termequivPToTick2aux {lu}{c} x Q termequivQ (inj2 t) = tnode (terc t)

termequivPToTick∞3 : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process∞ ∞ c)
  (termequivP : TerminateEquivalent∞ x P)
  → BisimForNextP (termequivPToTick∞1 {lu}{c} x P termequivP) (inj2 x)
termequivPToTick∞3 {lu}{c} x P termequivP = termequivPToTick3 {lu}{c} x (forcep P) termequivP

termequivPToTick3 : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process ∞ {lu} c)
  (termequivP : TerminateEquivalent x P)
  → BisimForNextP (termequivPToTick1 {lu}{c} x P termequivP) (inj2 x)
termequivPToTick3 {lu}{c} x .(terminate x) termeqterm = refl
termequivPToTick3 {lu}{c} x .(node Q) (termeqnode {Q} termequivQ) = termequivPToTick3aux {lu}

termequivPToTick3aux : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(Q : Process+ ∞ c)
  (termequivQ : TerminateEquivalent+ x Q)
  (hastauortick : ChoiceSet (I Q) ⊔ ChoiceSet (T Q) )
  → BisimForNextP (termequivPToTick1aux {lu}{c} x Q termequivQ hastauortick)
termequivPToTick3aux {lu}{c} x Q termequivQ (inj1 ip)
  = termequivPToTick∞3 {lu}{c} x (PI Q ip) (onlyIntChoice termequivQ ip)
termequivPToTick3aux {lu}{c} x Q termequivQ (inj2 t) rewrite termIsa termequivQ t = refl

```

```

termequivPToTick $\infty_1'$  : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process $\infty$   $\infty$  c)
  (termequivP : TerminateEquivalent $\infty$  x P)
  → Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
termequivPToTick $\infty_1'$  {lu}{c} x P termequivP = termequivPToTick $_1'$  x (forcep P) termequivP

termequivPToTick $_1'$  : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process  $\infty$  {lu} c)
  (termequivP : TerminateEquivalent x P)
  → Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
termequivPToTick $_1'$  {lu}{c} x (terminate .x) termeqterm = inj $_1$  (terminate x)
termequivPToTick $_1'$  {lu}{c} x (node Q) (termeqnode termequivQ)
  = termequivPToTick $_1$ aux'' x Q termequivQ (hasTauOrTick termequivQ)

termequivPToTick $_1$ aux'' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(Q : Process+  $\infty$  c)
  (termequivQ : TerminateEquivalent+ x Q)
  (hastauortick : ChoiceSet (I Q)  $\uplus$  ChoiceSet (T Q) )
  → Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
termequivPToTick $_1$ aux'' {lu}{c} x Q termequivQ (inj $_1$  ip)
  = termequivPToTick $\infty_1'$  {lu}{c} x (PI Q ip) (onlyIntChoice termequivQ ip)
termequivPToTick $_1$ aux'' {lu}{c} x Q termequivQ (inj $_2$  t) = inj $_2$  (PT Q t)

termequivPToTick $_2'$  : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process  $\infty$  {lu} c)
  (termequivP : TerminateEquivalent x P)
  → TrP [] (termequivPToTick $_1'$  {lu}{c} x P termequivP) P
termequivPToTick $_2'$  {lu}{c} x .(terminate x) termeqterm = empty x
termequivPToTick $_2'$  {lu}{c} x .(node Q) (termeqnode {Q} termequivQ)
  = termequivPToTick $_2$ aux' {lu}{c} x Q termequivQ (hasTauOrTick termequivQ)

termequivPToTick $_2$ aux' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(Q : Process+  $\infty$  c)
  (termequivQ : TerminateEquivalent+ x Q)
  (hastauortick : ChoiceSet (I Q)  $\uplus$  ChoiceSet (T Q) )
  → TrP [] (termequivPToTick $_1$ aux'' {lu}{c} x Q termequivQ hastauortick)
termequivPToTick $_2$ aux' x Q termequivQ (inj $_1$  x $_1$ )
  = tnode (intc [] (termequivPToTick $_1$ aux'' x Q termequivQ (inj $_1$  x $_1$ )) x $_1$ 
    (termequivPToTick $\infty_2'$  x (PI Q x $_1$ ) (onlyIntChoice termequivQ x $_1$ )))
termequivPToTick $_2$ aux' x Q termequivQ (inj $_2$  t) = tnode (terc t)

```

```

termequivPToTick $\infty_2$ ' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process $\infty$   $\infty$  c)
  (termequivP : TerminateEquivalent $\infty$  x P)
  → TrP $\infty$  [] (termequivPToTick $\infty_1$ ' {lu}{c} x P termequivP) P
termequivPToTick $\infty_2$ ' {lu}{c} x P termequivP = termequivPToTick $\infty_2$ ' {lu}{c} x (forceP P) termequivP

```

```

termequivPToTick $\infty_3$ ' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process $\infty$   $\infty$  c)
  (termequivP : TerminateEquivalent $\infty$  x P)
  → BisimForNextP (termequivPToTick $\infty_1$ ' {lu}{c} x P termequivP) (inj $_1$  (termequivPToTick $\infty_2$ ' {lu}{c} x P termequivP))
termequivPToTick $\infty_3$ ' {lu}{c} x P termequivP = termequivPToTick $\infty_3$ ' x (forceP P) termequivP

```

```

termequivPToTick $_3$ ' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(P : Process  $\infty$  {lu} c)
  (termequivP : TerminateEquivalent x P)
  → BisimForNextP (termequivPToTick $_1$ ' {lu}{c} x P termequivP) (inj $_1$  (termequivPToTick $_2$ ' {lu}{c} x P termequivP))
termequivPToTick $_3$ ' {lu}{c} x .(terminate x) termequivP = eqterminate termequivP
termequivPToTick $_3$ ' {lu}{c} x .(node Q) (termequivNode {Q} termequivQ)
  = termequivPToTick $_3$ aux' x Q termequivQ (hasTauOrTick termequivQ)

```

```

termequivPToTick $_3$ aux' : {lu : LUniv}{c : Choice}(x : ChoiceSet c)(Q : Process+  $\infty$  c)
  (termequivQ : TerminateEquivalent+ x Q)
  (hastauortick : ChoiceSet (I Q)  $\uplus$  ChoiceSet (T Q))
  → BisimForNextP (termequivPToTick $_1$ aux' {lu}{c} x Q termequivQ hastauortick)
termequivPToTick $_3$ aux' {lu}{c} x Q termequivQ (inj $_1$  ip)
  = termequivPToTick $\infty_3$ ' {lu}{c} x (PI Q ip) (onlyIntChoice termequivQ ip)
termequivPToTick $_3$ aux' {lu}{c} x Q termequivQ (inj $_2$  t) rewrite (termIsa termequivQ t) = termequivPToTick $\infty_3$ ' {lu}{c} x Q termequivQ

```

mutual

```

termEquivPr+ : {lu : LUniv}(c : Choice)(l : List (Label lu))(y : ChoiceSet c)(P' : Process+  $\infty$  c)
  (a : ChoiceSet c) (terequivP : TerminateEquivalent+ a P')
  (tr : TrP+ l (inj $_2$  y) P')
  → a  $\equiv$  y
termEquivPr+ c .(Lab P' x :: l) y P' a terequivP (extc l .(inj $_2$  y) x x $_1$ )
  =  $\perp$ -elim (noExtChoice terequivP x)
termEquivPr+ c l y P' a terequivP (intc .l .(inj $_2$  y) x tr)
  = termEquivPr c l y P' a (termequivNode terequivP) (tnode (intc l (inj $_2$  y) x tr))

```

$$\text{termEquivPr}+ c . [] . (\text{PT } P' x) P' a \text{ terequivP } (\text{terc } x) = \text{termIsa } \text{ terequivP } x$$

$$\begin{aligned} \text{termEquivPr} : \{lu : \text{LUniv}\} (c : \text{Choice}) (l : \text{List } (\text{Label } lu)) (y : \text{ChoiceSet } c) (P' : \text{Process}+ c) \\ (a : \text{ChoiceSet } c) (\text{terequivP} : \text{TerminateEquivalent } a (\text{node } P')) \\ (tr : \text{TrP } l (\text{inj}_2 y) (\text{node } P')) \\ \rightarrow a \equiv y \end{aligned}$$

$$\begin{aligned} \text{termEquivPr } c . (\text{Lab } P' x :: l) y P' a (\text{termeqnode } \text{ terequivP}) (\text{tnode } (\text{extc } l . (\text{inj}_2 y) x x_1)) \\ = \perp\text{-elim } (\text{noExtChoice } \text{ terequivP } x) \end{aligned}$$

$$\begin{aligned} \text{termEquivPr } c l y P' a (\text{termeqnode } \text{ terequivP}) (\text{tnode } (\text{intc } l . (\text{inj}_2 y) x x_1)) \\ = \text{termEquivPr} \infty c l y (\text{PI } P' x) a (\text{onlyIntChoice } \text{ terequivP } x) x_1 \end{aligned}$$

$$\text{termEquivPr } c . [] . (\text{PT } P' x) P' a (\text{termeqnode } \text{ terequivP}) (\text{tnode } (\text{terc } x)) = \text{termIsa } \text{ terequivP}$$

$$\begin{aligned} \text{termEquivPr} \infty : \{lu : \text{LUniv}\} (c : \text{Choice}) (l : \text{List } (\text{Label } lu)) (y : \text{ChoiceSet } c) (P' : \text{Process} \infty) \\ (a : \text{ChoiceSet } c) (\text{terequivP} : \text{TerminateEquivalent } a (\text{force } P')) \\ (tr : \text{TrP} \infty l (\text{inj}_2 y) P') \\ \rightarrow a \equiv y \end{aligned}$$

$$\text{termEquivPr} \infty c l y P' a \text{ terequivP } tr = \text{termEquivPr}' c l y (\text{forcep } P') a \text{ terequivP } tr$$

$$\begin{aligned} \text{termEquivPr}' : \{lu : \text{LUniv}\} (c : \text{Choice}) (l : \text{List } (\text{Label } lu)) (y : \text{ChoiceSet } c) (P' : \text{Process} \infty) \\ (a : \text{ChoiceSet } c) (\text{terequivP} : \text{TerminateEquivalent } a P') \\ (tr : \text{TrP } l (\text{inj}_2 y) P') \\ \rightarrow a \equiv y \end{aligned}$$

$$\text{termEquivPr}' c . [] y . (\text{terminate } y) . y \text{ termeqterm } (\text{ter } y) = \text{refl}$$

$$\text{termEquivPr}' c l y . (\text{node } P') a \text{ terequivP } (\text{tnode } \{.l\} \{.(\text{inj}_2 y)\} \{P'\} tr) = \text{termEquivPr } c l$$

mutual

$$\begin{aligned} \text{termEquivPreservedByTrace}+ : \{lu : \text{LUniv}\} (c : \text{Choice}) (l : \text{List } (\text{Label } lu)) (P' : \text{Process}+ c) \\ (Q : \text{Process} \infty \{lu\} c) (a : \text{ChoiceSet } c) \\ (\text{terequivP} : \text{TerminateEquivalent}+ a P') \\ (tr : \text{TrP}+ l (\text{inj}_1 Q) P') \\ \rightarrow \text{TerminateEquivalent } a Q \end{aligned}$$

$$\text{termEquivPreservedByTrace}+ c . [] P' . (\text{node } P') a \text{ terequivP } \text{empty} = \text{termeqnode } \text{ terequivP}$$

$$\text{termEquivPreservedByTrace}+ c . (\text{Lab } P' x :: l) P' Q a \text{ terequivP } (\text{extc } l . (\text{inj}_1 Q) x x_1) = \perp\text{-elim}$$

$$\text{termEquivPreservedByTrace}+ c l P' Q a \text{ terequivP } (\text{intc } l . (\text{inj}_1 Q) x tr)$$

$$= \text{termEquivPreservedByTrace } c l P' Q a (\text{termeqnode } \text{ terequivP}) (\text{tnode } (\text{intc } l (\text{inj}_1 Q) x$$

$$\text{termEquivPreservedByTrace} : \{lu : \text{LUniv}\} (c : \text{Choice}) (l : \text{List } (\text{Label } lu)) (P' : \text{Process}+ c)$$

```

      (Q : Process ∞ {lu} c)(a : ChoiceSet c)
      (terequivP : TerminateEquivalent a (node P'))
      (tr : TrP l (inj1 Q) (node P'))
      → TerminateEquivalent a Q
termEquivPreservedByTrace c .[] P' .(node P') a (termeqnode terequivP) (tnode empty) = termeqnode
termEquivPreservedByTrace c .(Lab P' x :: l) P' Q a (termeqnode terequivP)
      (tnode (extc l .(inj1 Q) x x1)) = ⊥-elim (noExtChoice t
termEquivPreservedByTrace c l P' Q a (termeqnode terequivP) (tnode (intc .l .(inj1 Q) x tr))
      = termEquivPreservedByTrace∞ c l (PI P' x) Q a (onlyIntChoice terequivP x) tr

termEquivPreservedByTrace∞ :      {lu : LUniv}{c : Choice}(l : List (Label lu))(P' : Process∞ ∞ c)
      (Q : Process ∞ {lu} c)(a : ChoiceSet c)
      (terequivP : TerminateEquivalent a (forceP P'))
      (tr : TrP∞ l (inj1 Q) P')
      → TerminateEquivalent a Q
termEquivPreservedByTrace∞ c l P' Q a terequivP tr = termEquivPreservedByTrace' c l (forceP P')

termEquivPreservedByTrace' :      {lu : LUniv}{c : Choice}(l : List (Label lu))(P' : Process ∞ {lu} c)
      (Q : Process ∞ {lu} c)(a : ChoiceSet c)
      (terequivP : TerminateEquivalent a P')
      (tr : TrP l (inj1 Q) P')
      → TerminateEquivalent a Q
termEquivPreservedByTrace' c .[] .(terminate x) .(terminate x) a terequivP (empty x) = terequivP
termEquivPreservedByTrace' c l .(node P') Q a terequivP (tnode {.l} {.inj1 Q}) {P'} x
      = termEquivPreservedByTrace c l P' Q a terequivP (tnode x)

```

mutual

--@BEGIN@bisimTraceTrPoneinf

```

bisimTraceTrP∞1 : {lu : LUniv}{c : Choice}(P P' : Process∞ ∞ c)
      (PP' : Bisimw∞ {∞} P P')(l : List (Label lu))
      (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
      (tr : TrP∞ l tick P')
      → Process ∞ {lu} c ⊔ ChoiceSet c
bisimTraceTrP∞1 P P' PP' l x tr = bisimTraceTrP1 (forceP P)
      (forceP P')
      (forceB PP')

```

$$l \ x \ tr$$

--@END

--@BEGIN@bisimTraceTrPtwoinf

```

bisimTraceTrP $\infty_2$  : {lu : LUniv}{c : Choice}(P P' : Process $\infty$   $\infty$  c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')(l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP $\infty$  l tick P')
  → TrP $\infty$  l (bisimTraceTrP $\infty_1$  P P' PP' l tick tr) P
bisimTraceTrP $\infty_2$  P P' PP' l x tr = bisimTraceTrP $_2$  (forcep P)
  (forcep P')
  (forceB PP')
  l x tr

```

--@END

--@BEGIN@bisimTraceTrPthreeinf

```

bisimTraceTrP $\infty_3$  : {lu : LUniv}{c : Choice}(P P' : Process $\infty$   $\infty$  c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')(l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP $\infty$  l tick P')
  → BisimForNextP (bisimTraceTrP $\infty_1$  P P' PP' l tick tr) tick
bisimTraceTrP $\infty_3$  P P' PP' l x tr = bisimTraceTrP $_3$  (forcep P)
  (forcep P')
  (forceB PP')
  l x tr

```

--@END

--@BEGIN@bisimTraceTrPone

```

bisimTraceTrP $_1$  : {lu : LUniv}{c : Choice}(P P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  (l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP l tick P')
  → Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
bisimTraceTrP $_1$  .(terminate y) (terminate x)(eqterminate {y} x $_1$ )

```

```

      .[] .(inj2 x) (ter .x) = inj2 y
bisimTraceTrP1 P (terminate x) (eqterminater termequivP)
      .[] .(inj2 x) (ter .x) =
      termequivPToTick1 x P termequivP
bisimTraceTrP1 .(terminate y) (terminate x)
      (eqterminate {y} x1) .[]
      .(inj1 (terminate x))(empty .x)
      = inj1 (terminate y)
bisimTraceTrP1 P (terminate x) (eqterminater termequivP) .[]
      .(inj1 (terminate x)) (empty .x) =
      termequivPToTick1' x P termequivP
bisimTraceTrP1 .(terminate a) (node P')
      (eqterminate {a} nodexTerEquiv)
      l (inj1 x) (tnode tr) = inj1 (terminate a)
bisimTraceTrP1 .(terminate a) (node P')
      (eqterminate {a} nodexTerEquiv)
      l (inj2 y) (tnode tr) = inj2 a
bisimTraceTrP1 .(node P) (node P')
      (eqnode {P} PP') l tick
      (tnode tr) = bisimTraceTrP1+ P P' PP' l tick tr

```

```
--@END
```

```
--@BEGIN@bisimTraceTrPtwo
```

```

bisimTraceTrP2 : {lu : LUniv}{c : Choice}
      (P P' : Process ∞ {lu} c)
      (PP' : Bisimw {∞} P P')
      (l : List (Label lu))
      (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
      (tr : TrP l tick P')
  → TrP l (bisimTraceTrP1 P P' PP' l tick tr) P
bisimTraceTrP2 .(terminate y) .(terminate x)
      (eqterminate {y} x1)
      .[] .(inj2 x) (ter x) = ter y
bisimTraceTrP2 P .(terminate x)
      (eqterminater termequivP)
      .[] .(inj2 x)
      (ter x) = termequivPToTick2 x P termequivP
bisimTraceTrP2 .(terminate y) .(terminate x)
      (eqterminate {y} x1)

```

```

bisimTraceTrP2 .[] .(inj1 (terminate x))(empty x) = empty y
bisimTraceTrP2 P .(terminate x)
  (eqterminater termequivP) .[]
bisimTraceTrP2 .(inj1 (terminate x)) (empty x) =
  termequivPToTick2' x P termequivP
bisimTraceTrP2 {lu} {c} .(terminate a) .(node P')
  (eqterminate {a} termequivP) l
  (inj1 x) (tnode {l} {(inj1 x)} {P'} tr)
  rewrite
  (lemmaTraceTerminationEquivalentEmpty+'
   c l P' (inj1 x) a termequivP tr) = empty a
bisimTraceTrP2 {lu}{c} .(terminate a) .(node P')
  (eqterminate {a} termequivP) l
  (inj2 y) (tnode {l} {(inj2 y)} {P'} tr)
  rewrite
  (lemmaTraceTerminationEquivalentEmpty+' c l P'
   (inj2 y) a termequivP tr) = ter a
bisimTraceTrP2 {lu}{c} .(node P) .(node P')
  (eqnode {P} {P'} PP') l tick
  (tnode tr) =
  tnode (bisimTraceTrP2+ P P' PP' l tick tr)

--@END

--@BEGIN@bisimTraceTrPthree

bisimTraceTrP3 : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (l : List (Label lu))
  (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  (tr : TrP l tick P')
  → BisimForNextP
  (bisimTraceTrP1 P P' PP' l tick tr) tick
bisimTraceTrP3 .(terminate x) (terminate x)
  (eqterminate {x} termeqterm)
bisimTraceTrP3 .[] .(inj2 x) (ter .x) = refl
bisimTraceTrP3 P (terminate x)
  (eqterminater termequivP) .[] .(inj2 x)
  (ter .x) = termequivPToTick3 x P termequivP
bisimTraceTrP3 .(terminate x) (terminate x)

```

```

      (eqterminate {x} termeqterm) .[]
    .(inj1 (terminate x))
      (empty x) = eqterminate termeqterm
bisimTraceTrP3    P (terminate x)
      (eqterminater termequivP) .[]
    .(inj1 (terminate x)) (empty x) =
bisimTraceTrP3    termequivPToTick3' x P termequivP
      {lu}{c} .(terminate a) (node P')
      (eqterminate {a} (termeqnode terequivP))
      l (inj1 Q) (tnode tr) = eqterminate
bisimTraceTrP3    (termEquivPreservedByTrace+
      c l P' Q a terequivP tr)
      {lu}{c} .(terminate a) (node P')
      (eqterminate {a} (termeqnode terequivP))
bisimTraceTrP3    l (inj2 y) (tnode tr) =
      termEquivPr+ c l y P' a terequivP tr
bisimTraceTrP3    .(node P) (node P')
      (eqnode {P} PP') l tick
      (tnode tr) =
bisimTraceTrP3+ P P' PP' l tick tr

```

```
--@END
```

```
--@BEGIN@bisimTraceTrPoneplus
```

```

bisimTraceTrP1+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ c)
  (PP' : Bisimw+ {∞} P P')
  (l : List (Label lu))
  (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  (tr : TrP+ l tick P')
  → Process ∞ {lu} c ⊔ ChoiceSet c
bisimTraceTrP1+ P P' PP' .[] .(inj1 (node P')) empty =
  (inj1 (node P))
bisimTraceTrP1+ {lu}{c} P P' PP'
  .(Lab P' x :: l) tick (extc l .tick x tr) =
  bisimTraceTrP∞1 R R' RR' l tick tr
module bisimETraceTrP1+auxmodule where
  R' : Process∞ ∞ c
  R' = PE P' x

```



```

R : Process $\infty$   $\infty$  c
R = bisimEP'r PP' x

RR' : Bisimw $\infty$  { $\infty$ } R R'
RR' = bisimEnextr PP' x

bisimTraceTrP1+ {lu}{c} P P' PP' l tick
  (intc .l .tick x x1) =
    bisimTraceTrP $\infty$ 1 R R' RR' l tick x1
module bisimTraceTrP1+auxmodule where
  R' : Process $\infty$   $\infty$  c
  R' = PI P' x

R : Process $\infty$   $\infty$  c
R = bisimIP'r PP' x

RR' : Bisimw $\infty$  { $\infty$ } R R'
RR' = bisimInextr PP' x

bisimTraceTrP1+ P P' PP' .[] .(inj2 (PT P' x))
  (terc x) = inj2 (PT P' x)

--@END

--@BEGIN@bisimTraceTrPtowplus

bisimTraceTrP2+ : {lu : LUniv}{c : Choice}
  (P P' : Process+  $\infty$  c)
  (PP' : Bisimw+ { $\infty$ } P P')
  (l : List (Label lu))
  (tick : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (tr : TrP+ l tick P')
  → TrP+ l (bisimTraceTrP1+ P P' PP' l tick tr) P
bisimTraceTrP2+ P P' PP' .[] .(inj1 (node P')) empty = empty
bisimTraceTrP2+ {lu}{c} P P' PP' .(Lab P' x :: l)
  tick (extc l .tick x tr1') = tr
module bisimETraceTrP2+auxmodule where
  R' : Process $\infty$   $\infty$  c
  R' = PE P' x

R : Process $\infty$   $\infty$  c

```



```

R = bisimEP'r PP' x

RR' : Bisimw∞ {∞} R R'
RR' = bisimEnextr PP' x

Rhat : Process ∞ {lu} c ⊔ ChoiceSet c
Rhat = bisimTraceTrP1 (forcep R)
      (forcep (PE P' x)) (forceB RR') l tick tr1'
tr1 : P →+*[ Lab P' x :: [] ] (forcep R)
tr1 = bisimEtrr PP' x

tr2 : TrP∞ {lu}{c} l Rhat R
tr2 = bisimTraceTrP2 (forcep R) (forcep (PE P' x))
      (forceB RR') l tick tr1'

tr : TrP+ (Lab P' x :: l) Rhat P
tr = trPAppendTrw+ c P (forcep R) (Lab P' x :: []) l
      Rhat tr1 tr2

bisimTraceTrP2+ {lu}{c} P P' PP' l
  tick (intc .l .tick x tr2') = tr
module bisimlTraceTrP2+auxmodule where
  R' : Process∞ ∞ c
  R' = PI P' x

  R : Process∞ ∞ c
  R = bisimIP'r PP' x

  RR' : Bisimw∞ {∞} R R'
  RR' = bisimInextr PP' x

  Rhat : Process ∞ {lu} c ⊔ ChoiceSet c
  Rhat = bisimTraceTrP1 (forcep (bisimIP'r PP' x))
        (forcep (PI P' x))
        (forceB (bisimInextr PP' x)) l tick tr2'

  tr1 : P →+*[ [] ] (forcep R)
  tr1 = bisimltrr PP' x

  tr2 : TrP l Rhat (forcep R)
  tr2 = bisimTraceTrP2 (forcep R)

```

```

    (forcep (PI P' x))
    (forceB (bisimInextr PP' x)) l tick tr₂'

tr :    TrP+ l Rhat P
tr =    trPAppendTrw+ c P
        (forcep R) [] l Rhat tr₁ tr₂

bisimTraceTrP₂+ {lu}{c} P P' PP' .[]
               .(inj₂ (PT P' x)) (terc x) = tr₁
where
  tr₁ : TrP+ [] (inj₂ (PT P' x)) P
  tr₁ = bisimTtrr PP' x

--@END

--@BEGIN@bisimTraceTrPthreeplus

bisimTraceTrP₃+ : {lu : LUniv}{c : Choice}
                 (P P' : Process+ ∞ c)
                 (PP' : Bisimw+ {∞} P P')
                 (l : List (Label lu))
                 (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
                 (tr : TrP+ l tick P')
→ BisimForNextP
  (bisimTraceTrP₁+ P P' PP' l tick tr) tick
bisimTraceTrP₃+ P P' PP' .[] .(inj₁ (node P'))
empty = eqnode PP'
bisimTraceTrP₃+ P P' PP'
               .(Lab P' x :: l) (inj₁ x₁)
               (extc l .(inj₁ x₁) x x₂) =
bisimTraceTrP₃ (forcep
  (bisimETraceTrP₁+auxmodule.R P' l x
    (inj₁ x₁) P PP' x₂)) (forcep (PE P' x))
  (forceB (bisimETraceTrP₁+auxmodule.RR'
    P' l x (inj₁ x₁) P PP' x₂)) l
    (inj₁ x₁) x₂
bisimTraceTrP₃+ {lu}{c} P P' PP' .(Lab P' x :: l) (inj₂ y)
               (extc l .(inj₂ y) x x₁) =
bisimTraceTrP₃
  (forcep (bisimETraceTrP₁+auxmodule.R P' l x

```

```

      (inj2 y) P PP' x1) (forcep (PE P' x))
      (forceB (bisimETraceTrP1+auxmodule.RR'
      P' l x (inj2 y) P PP' x1) l (inj2 y) x1
      {lu}{c} P P' PP' l (inj1 x)
      (intc .l .(inj1 x) x1 x2) =
      bisimTraceTrP3
      (forcep (bisimITraceTrP1+auxmodule.R l
      (inj1 x) P' P PP' x1 x2))
      (forcep (PI P' x1))
      (forceB (bisimITraceTrP1+auxmodule.RR' l
      (inj1 x) P' P PP' x1 x2)) l (inj1 x) x2
      {lu}{c} P P' PP' l (inj2 y) (intc .l .(inj2 y) x x1) =
      bisimTraceTrP3
      (forcep (bisimITraceTrP1+auxmodule.R l (inj2 y) P'
      P PP' x x1))
      (forcep (PI P' x))
      (forceB (bisimITraceTrP1+auxmodule.RR' l (inj2 y)
      P' P PP' x x1)) l (inj2 y) x1
      bisimTraceTrP3+ {lu}{c} P P' PP' .[] .(inj2 (PT P' x)) (terc x) = refl

```

```
--@END
```

```
{- the following lemma is false if tickincluded = false-}
```

```
mutual
```

```

lemmaDrefusalStableNotTermequiv : {lu : LUniv}{c : Choice}(Q : Process ∞ {lu} c)
  (X : (Label lu) → Bool)
  (dref : DRefusal Q true X)
  (stab : stable Q)
  (y : ChoiceSet c)
  (termequivQ : TerminateEquivalent y Q)
  → ⊥

```

```
lemmaDrefusalStableNotTermequiv (terminate x) X dref stab y termequiv = dref tt
```

```
lemmaDrefusalStableNotTermequiv (node Q) X (drefusal noextChInX noTerm)
```

```
stab y (termegnode terequivP) = hasTauOrTickGivesBot hasTauOrTickNoTa
```

```
where
```

```
hasTauOrTickNoTau' : ChoiceSet (I Q) ⊕
```

```
¬ (ChoiceSet (I Q)) × ChoiceSet (T Q)
```

`hasTauOrTickNoTau' = hasTauOrTickNoTau terequivP`

`hasTauOrTickGivesBot : ChoiceSet (I Q) ⊔ ¬ (ChoiceSet (I Q)) × ChoiceSet (T`
`hasTauOrTickGivesBot (inj1 x) = stabToNoInternal+ Q stab x --stab x`
`hasTauOrTickGivesBot (inj2 (noti „ t)) = noTerm _ t`

`lemmaDivNotTermequiv : {lu : LUniv}{c : Choice}(Q : Process ∞ {lu} c)`
`(divQ : DivergentProcess ∞ {lu} c Q)`
`(x : ChoiceSet c)`
`(termequivQ : TerminateEquivalent x Q)`
`→ ⊥`

`lemmaDivNotTermequiv .(node P) (div P (div+ int divP)) x (termeqnode terequivP)`
`= lemmaDivNotTermequiv (forcep (PI P int)) (forcediv divP) x (onlyIntChoice terequivP in`

A.6 bisimilarity.agda

`--@PREFIX@bisim`

`module bisimilarity where`

`open import process`
`open import choiceSetU`
`open import labelUniv`
`open import Size`
`open import Relation.Binary.PropositionalEquality`
`open import Data.Unit.Base`
`open import Data.Empty`
`open import Data.List`
`open import Data.Sum`
`open import TraceWithNextProcess`
`open import auxData`
`open import dataAuxFunction`
`open import fdi`

```

lemmaorcross1 : {A B C : Set}(ab : A ⊕ (B × C)) → A ⊕ C
lemmaorcross1 (inj1 a) = inj1 a
lemmaorcross1 (inj2 (b , c)) = inj2 c

lemmaorcross2 : {A B C : Set}(ab : A ⊕ (B × C)) → A ⊕ B
lemmaorcross2 (inj1 a) = inj1 a
lemmaorcross2 (inj2 (b , c)) = inj2 b

mutual

--@BEGIN@bisimsDefinf

record Bisims∞ {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ {lu} c) : Set where
  coinductive
  field
    forceB : {j : Size< i} → Bisims {j} {lu}(forceP P) (forceP P')

--@END

--@BEGIN@bisimsDef

data Bisims {i : Size}{lu : LUniv}{c : Choice} :
  (P P' : Process ∞ {lu} c) → Set where
  eqterminate : { a : ChoiceSet c}
    → Bisims {i} (terminate a) (terminate a)
  eqnode      : {Q Q' : Process+ ∞ {lu} c} → Bisims+ {i} Q Q'
    → Bisims {i} (node Q) (node Q')

--@END

--@BEGIN@bisimsDefPlus

record Bisims+ {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c) : Set where
  coinductive

```

field

```

bisim2E      : (e : ChoiceSet (E P)) → ChoiceSet (E P')
bisimELab    : (e : ChoiceSet (E P))
               → Lab P e ≡ Lab P' (bisim2E e)

bisimENext   : (e : ChoiceSet (E P))
               → Bisims∞ {i} (PE P e) (PE P' (bisim2E e))

bisim2I      : (int1 : ChoiceSet (I P)) → ChoiceSet (I P')
bisimINext   : (int1 : ChoiceSet (I P))
               → Bisims∞ {i} (PI P int1) (PI P' (bisim2I int1))

bisim2T      : (t : ChoiceSet (T P)) → ChoiceSet (T P')
bisim2TEq    : (t : ChoiceSet (T P))
               → PT P t ≡ PT P' (bisim2T t)

bisim2Er     : (e : ChoiceSet (E P')) → ChoiceSet (E P)
bisimELabr   : (e : ChoiceSet (E P'))
               → Lab P' e ≡ Lab P (bisim2Er e)

bisimENextr  : (e : ChoiceSet (E P'))
               → Bisims∞ {i} (PE P (bisim2Er e)) (PE P' e)

bisim2Irr    : (int1 : ChoiceSet (I P')) → ChoiceSet (I P)
bisimINextr  : (int1 : ChoiceSet (I P'))
               → Bisims∞ {i} (PI P (bisim2Irr int1)) (PI P' int1)

bisim2Tr     : (t : ChoiceSet (T P')) → ChoiceSet (T P)
bisim2TEqr   : (t : ChoiceSet (T P'))
               → PT P' t ≡ PT P (bisim2Tr t)

```

--@END

open Bisims∞ public

open Bisims+ public

mutual

```

swapChoiceSetss : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
                  (PP' : Bisims+ {i} P P')
                  (p : ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P)))
                  → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))

swapChoiceSetss {i} {c} P P' PP' (inj1 x) = inj1 (bisim2I PP' x)
swapChoiceSetss {i} {c} P P' PP' (inj2 y) = inj2 (λ y' → y (bisim2Irr PP' y'))

```

```

swapChoiceSetssr : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)

```

```

      (PP' : Bisims+ {i} P P')
      (p : ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P')))
      → ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P))
swapChoiceSetssr {i} {c} P P' PP' (inj1 x) = inj1 (bisim2lr PP' x)
swapChoiceSetssr {i} {c} P P' PP' (inj2 y) = inj2 (λ y' → y (bisim2l PP' y'))

mutual
--@BEGIN@Nondivergent

record NonDivergent∞ {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c) : Set where
  inductive
  field
    forceND : {j : Size< i} → NonDivergent {j} (forcep P)

NonDivergent : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c) → Set
NonDivergent (terminate x) = ⊤
NonDivergent {i} (node Q) = NonDivergent+ {i} Q

data NonDivergent+ {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c) : Set where
  nondiv : ((int1 : ChoiceSet (I P)) → NonDivergent∞ {i} (PI P int1))
    → (chemptyornot : ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P)))
    → NonDivergent+ {i} P

--@END

open NonDivergent∞ public

mutual

--@BEGIN@TerminateEquivalentinf

TerminateEquivalent∞ : {lu : LUniv}{c : Choice}(a : ChoiceSet c)
  (P : Process∞ ∞ {lu} c) → Set
TerminateEquivalent∞ a P = TerminateEquivalent a (forcep P)

--@END

```

```

--@BEGIN@TerminateEquivalent

data TerminateEquivalent {lu : LUniv}{c : Choice}(a : ChoiceSet c)
  : (P : Process  $\infty$  {lu} c)  $\rightarrow$  Set where
  termeqterm : TerminateEquivalent a (terminate a)
  termeqnode : {P : Process+  $\infty$  {lu} c}
    (terequivP : TerminateEquivalent+ a P)
     $\rightarrow$  TerminateEquivalent a (node P)

--@END

--@BEGIN@TerminateEquivalentplus

record TerminateEquivalent+ {lu : LUniv}{c : Choice}(a : ChoiceSet c)
  (P : Process+  $\infty$  {lu} c) : Set where
  inductive
  field
    noExtChoice      : (e : ChoiceSet (E P))  $\rightarrow$   $\perp$ 
    onlyIntChoice    : (i : ChoiceSet (I P))
       $\rightarrow$  TerminateEquivalent $\infty$  a (PI P i)
    termIsa          : (t : ChoiceSet (T P))  $\rightarrow$  a  $\equiv$  PT P t
    hasTauOrTickNoTau : ChoiceSet (I P)  $\uplus$ 
      ( $\neg$  (ChoiceSet (I P))  $\times$  ChoiceSet (T P))

--@END

open TerminateEquivalent+ public

hasTauOrTick : {lu : LUniv}{c : Choice}
  {a : ChoiceSet c}
  {P : Process+  $\infty$  {lu} c}
  (termequiv : TerminateEquivalent+ a P)
   $\rightarrow$  ChoiceSet (I P)  $\uplus$  ChoiceSet (T P)
hasTauOrTick termequiv = lemmaorcross1 (hasTauOrTickNoTau termequiv)

hasTauOrNotTau : {lu : LUniv}{c : Choice}
  {a : ChoiceSet c}
  {P : Process+  $\infty$  {lu} c}
  (termequiv : TerminateEquivalent+ a P)
   $\rightarrow$  ChoiceSet (I P)  $\uplus$   $\neg$  (ChoiceSet (I P))
hasTauOrNotTau termequiv = lemmaorcross2 (hasTauOrTickNoTau termequiv)

```

mutual

--@BEGIN@bisimDefinf

```
record Bisimw $\infty$  {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process $\infty$   $\infty$  {lu} c) : Set where
  coinductive
  field
    forceB : {j : Size < i}  $\rightarrow$  Bisimw {j}{lu} (forceP P) (forceP P')
```

--@END

--@BEGIN@bisimDef

```
data Bisimw {i : Size}{lu : LUniv}{c : Choice}
  : (P P' : Process $\infty$  {lu} c)  $\rightarrow$  Set where
  eqterminate : {a : ChoiceSet c}  $\rightarrow$  {P' : Process $\infty$  {lu} c}
    (terequiv : TerminateEquivalent a P')
     $\rightarrow$  Bisimw {i} (terminate a) P'
  eqterminater : {a : ChoiceSet c}  $\rightarrow$  {P : Process $\infty$  {lu} c}
    (terequiv : TerminateEquivalent a P)
     $\rightarrow$  Bisimw {i} P (terminate a)
  eqnode : {Q Q' : Process+  $\infty$  {lu} c}
    (bisimQQ' : Bisimw+ {i} Q Q')
     $\rightarrow$  Bisimw {i} (node Q) (node Q')
```

--@END

--@BEGIN@bisimDefPlus

```
record Bisimw+ {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process+  $\infty$  {lu} c) : Set where
  coinductive
  field
    bisimdiv : DivergentProcess+ i c P  $\rightarrow$  DivergentProcess+ i c P'
    nondiv+ : NonDivergent+ {i} P  $\rightarrow$  NonDivergent+ {i} P'
    bisimEP' : (e : ChoiceSet (E P))  $\rightarrow$  Process $\infty$   $\infty$  {lu} c
```

```

bisimEtr      : (e : ChoiceSet (E P))
               → P' →+*[ Lab P e :: [] ] (forcep (bisimEP' e))
bisimEnext    : (e : ChoiceSet (E P))
               → Bisimw∞ {i} (PE P e) (bisimEP' e)
bisimIP'      : (int1 : ChoiceSet (I P)) → Process∞ ∞ {lu} c
bisimltr      : (int1 : ChoiceSet (I P))
               → P' →+*[ [] ] (forcep (bisimIP' int1))
bisimlnext    : (int1 : ChoiceSet (I P))
               → Bisimw∞ {i} (PI P int1) (bisimIP' int1)
bisimTtr      : (t : ChoiceSet (T P)) → TrP+ [] (inj2 (PT P t)) P'
bisimdivr     : DivergentProcess+ i c P' → DivergentProcess+ i c P
nondiv+r      : NonDivergent+ {i} P' → NonDivergent+ {i} P
bisimEP'r     : (e : ChoiceSet (E P')) → Process∞ ∞ {lu} c
bisimEtrr     : (e : ChoiceSet (E P'))
               → TrP+ (Lab P' e :: []) (inj1 (forcep (bisimEP'r e))) P
bisimEnexttr  : (e : ChoiceSet (E P'))
               → Bisimw∞ {i} (bisimEP'r e) (PE P' e)
bisimIP'r     : (int1 : ChoiceSet (I P')) → Process∞ ∞ {lu} c
bisimltrr     : (int1 : ChoiceSet (I P'))
               → TrP+ [] (inj1 (forcep (bisimIP'r int1))) P
bisimlnextr   : (int1 : ChoiceSet (I P'))
               → Bisimw∞ {i} (bisimIP'r int1) (PI P' int1)
bisimTtrr     : (t : ChoiceSet (T P')) → TrP+ [] (inj2 (PT P' t)) P

--@END

open Bisimw∞ public
open Bisimw+ public

```

A.7 bisimilarityProofs.agda

```
--@PREFIX@bisimilarityProofs
```

```
module bisimilarityProofs where
```

```
open import process
open import choiceSetU
open import labelUniv
```

```

open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool          renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable

mutual
  nondivImpliesIPorNotIP : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
    (nondiv : NonDivergent+ P)
    → ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P))
  nondivImpliesIPorNotIP {c} P
    (nondiv - chemptyornot) = chemptyornot

mutual

  swapChoiceSets : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
    (PP' : Bisimw+ {∞} P P')
    (nondiv : NonDivergent+ P)
    → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))
  swapChoiceSets {c} P P' PP' nond
    = nondivImpliesIPorNotIP P' (nondiv+ PP' nond)

```

$\text{stableImpliesNonDiv}\infty : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}\infty \infty \{lu\} c)$
 $(PS : \text{stable}\infty P) \rightarrow \text{NonDivergent}\infty P$
 $\text{forceND } (\text{stableImpliesNonDiv}\infty P PS) = \text{stableImpliesNonDiv } (\text{forcep } P) PS$

$\text{stableImpliesNonDiv} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process} \infty \{lu\} c)$
 $(PS : \text{stable } P)$
 $\rightarrow \text{NonDivergent } P$

$\text{stableImpliesNonDiv } (\text{terminate } x) PS = \text{tt}$
 $\text{stableImpliesNonDiv } (\text{node } x) PS = \text{stableImpliesNonDiv+ } x PS$

$\text{stableImpliesNonDiv+} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process+} \infty \{lu\} c)$
 $(PS : \text{stable+ } P) \rightarrow \text{NonDivergent+ } P$

$\text{stableImpliesNonDiv+ } P PS = \text{nondiv}$
 $(\text{stableImpliesNonDiv+aux } P PS) (\text{inj}_2 (\text{stabToNoInternal+ } P PS))$

$\text{stableImpliesNonDiv+aux} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process+} \infty \{lu\} c)$
 $(PS : \text{stable+ } P)(i : \text{ChoiceSet } (\text{I } P))$
 $\rightarrow \text{NonDivergent}\infty (\text{PI } P i)$

$\text{stableImpliesNonDiv+aux } P PS i = \perp\text{-elim } (\text{stabToNoInternal+ } P PS i)$

mutual

$\text{TerImpliesNotDivergentaux+} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process+} \infty \{lu\} c)$
 $(a : \text{ChoiceSet } c)$
 $(\text{terequiv} : \text{TerminateEquivalent+ } a P)$
 $\rightarrow \text{NonDivergent+ } P$

$\text{TerImpliesNotDivergentaux+ } c P a$
 $\text{terequiv} = \text{nondiv}$
 $(\lambda i \rightarrow \text{TerImpliesNotDivergentaux}\infty c$
 $(\text{PI } P i) a (\text{onlyIntChoice } \text{terequiv } i))$
 $(\text{hasTauOrNotTau } \text{terequiv}))$

$\text{TerImpliesNotDivergentaux} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process} \infty \{lu\} c)$
 $(a : \text{ChoiceSet } c)$
 $(\text{terequiv} : \text{TerminateEquivalent } a P)$
 $\rightarrow \text{NonDivergent } P$

$\text{TerImpliesNotDivergentaux } c (\text{terminate } x) a \text{terequiv} = \text{tt}$
 $\text{TerImpliesNotDivergentaux } c (\text{node } x) a (\text{termegnode } \text{terequiv} P)$
 $= \text{TerImpliesNotDivergentaux+ } c x a \text{terequiv} P$

$\text{TerImpliesNotDivergentaux}\infty : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process}\infty \infty \{lu\} c)$

```

      (a : ChoiceSet c)
      (terequiv : TerminateEquivalent a (forceP P))
      → NonDivergent∞ P
forceND (TerImpliesNotDivergentaux∞ c P a terequiv)
  = TerImpliesNotDivergentaux c (forceP P) a terequiv

```

```
--@BEGIN@bisimStableImpliesNotDivergent
```

```
mutual
```

```

bisimStableImpliesNotDivergent∞ : {lu : LUniv}(c : Choice)
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (PS' : stable∞ P')
  (nonDivP' : NonDivergent∞ P')
  → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞ c P P' PP' PS' nonDivP')
  = bisimStableImpliesNotDivergent c (forceP P)
    (forceP P')
    (forceB PP')
    PS' (forceND nonDivP')

```

```

bisimStableImpliesNotDivergent : {lu : LUniv}(c : Choice)
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (PS' : stable P')
  (nonDivP' : NonDivergent P')
  → NonDivergent P
bisimStableImpliesNotDivergent c (terminate x) P' PP' PS' nonDivP' = tt
bisimStableImpliesNotDivergent c (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' nonDivP' =
  TerImpliesNotDivergentaux c (node P)
    a ((termeqnode terequivP))

```

```

bisimStableImpliesNotDivergent c (node P) (node P') (eqnode PP') PS'
  nonDivP' = bisimStableImpliesNotDivergent+
    c P P' PP' PS' nonDivP'

```

```

bisimStableImpliesNotDivergent+ : {lu : LUniv}(c : Choice)
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (PS' : stable+ P')
  (nonDivP' : NonDivergent+ P')
  → NonDivergent+ P
bisimStableImpliesNotDivergent+ c P P' PP' PS' nonDivP'
  = nondiv+r PP' nonDivP'
--@END

```

```

--@BEGIN@nonDivBecomeStable

```

```

mutual

```

```

nonDivBecomeStable∞1 : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → Process ∞ {lu} c
nonDivBecomeStable∞1 c P nonDivP = nonDivBecomeStable1
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable∞2 : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → TrP∞ {lu} [(inj1
    (nonDivBecomeStable∞1 c P nonDivP)) P
nonDivBecomeStable∞2 c P nonDivP = nonDivBecomeStable2
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable∞3 : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → stableSch (nonDivBecomeStable∞1
    c P nonDivP)
nonDivBecomeStable∞3 c P nonDivP = nonDivBecomeStable3
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable+1 : {lu : LUniv}(c : Choice)
  (P : Process+ ∞ {lu} c)
  (nonDivP : NonDivergent+ P)
  → Process ∞ {lu} c

```

```

nonDivBecomeStable+1 c P (nondiv x (inj1 int)) =
    nonDivBecomeStable∞1
    c (PI P int) (x int)
nonDivBecomeStable+1 c P (nondiv x (inj2 stab)) = node P

nonDivBecomeStable+2 : {lu : LUniv}(c : Choice)
    (P : Process+ ∞ {lu} c)
    (nonDivP : NonDivergent+ P)
    → TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P
nonDivBecomeStable+2 c P (nondiv x (inj1 int)) = intc [] (inj1
    (nonDivBecomeStable+1 c P
    (nondiv x (inj1 int)))) int
    (nonDivBecomeStable∞2 c
    (PI P int) (x int))
nonDivBecomeStable+2 c P (nondiv x (inj2 stab)) = empty

nonDivBecomeStable+3 : {lu : LUniv}(c : Choice)
    (P : Process+ ∞ {lu} c)
    (nonDivP : NonDivergent+ P)
    → stableSch
    (nonDivBecomeStable+1 c P nonDivP)
nonDivBecomeStable+3 c P (nondiv x (inj1 int)) =
    nonDivBecomeStable∞3 c
    (PI P int) (x int)
nonDivBecomeStable+3 c P (nondiv x (inj2 stab)) = stab

nonDivBecomeStable1 : {lu : LUniv}(c : Choice)
    (P : Process ∞ {lu} c)
    (nonDivP : NonDivergent P)
    → Process ∞ {lu} c
nonDivBecomeStable1 c (terminate x) nonDivP = terminate x
nonDivBecomeStable1 c (node x) nonDivP =
    nonDivBecomeStable+1 c x nonDivP

nonDivBecomeStable2 : {lu : LUniv}(c : Choice)
    (P : Process ∞ {lu} c)
    (nonDivP : NonDivergent P)
    → TrP {lu} [] (inj1
    (nonDivBecomeStable1 c P nonDivP)) P

```

```

nonDivBecomeStable2 c (terminate x) nonDivP = empty x
nonDivBecomeStable2 c (node x) nonDivP
    = tnode (nonDivBecomeStable+2 c x nonDivP)

```

```

nonDivBecomeStable3 : {lu : LUniv}{c : Choice}
    (P : Process ∞ {lu} c)
    (nonDivP : NonDivergent P)
    → stableSch (nonDivBecomeStable1 c P nonDivP)
nonDivBecomeStable3 c (terminate x) nonDivP = _
nonDivBecomeStable3 c (node x) nonDivP = nonDivBecomeStable+3 c
    x nonDivP

```

```
--@END
```

mutual

```

nonDivBecomesStableBisimProof∞ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
    (nondiv' : NonDivergent∞ P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent∞ a P)
    → Bisimw (nonDivBecomeStable∞1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof∞ P nondiv' a terequivP =
    nonDivBecomesStableBisimProof (forceP P) (forceND nondiv') a terequivP

```

```

nonDivBecomesStableBisimProof : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
    (nondiv' : NonDivergent P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent a P)
    → Bisimw (nonDivBecomeStable1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof (terminate x) nondiv' .x termeqterm
    = BismwRef (terminate x)
nonDivBecomesStableBisimProof (node x) nondiv' a
    (termeqnode terequivP) =
    nonDivBecomesStableBisimProof+ x nondiv' a terequivP

```

```

nonDivBecomesStableBisimProof+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
    (nondiv' : NonDivergent+ P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent+ a P)

```

```

    → Bisimw (nonDivBecomeStable+1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof+ P (nondiv x (inj1 int)) a terequivP
    = nonDivBecomesStableBisimProof∞ (PI P int) (x int) a
      (onlyIntChoice terequivP int)
nonDivBecomesStableBisimProof+ P (nondiv x (inj2 stab)) a terequivP
    = eqterminater (termegnode terequivP)

```

mutual

```

emptyTrPtoQImpliesEq      : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
                           (PS' : stable P)(tr : TrP {lu} [] (inj1 P') P)
                           → P ≡ P'

emptyTrPtoQImpliesEq (terminate x) .(terminate x) pat (empty .x) = refl
emptyTrPtoQImpliesEq (node x) .(node x) PS' (tnode empty) = refl
emptyTrPtoQImpliesEq (node Q) P' PS'
  (tnode (intc .[] .(inj1 P') x1 x2)) = ⊥-elim (stabToNoInternal+ Q PS' x1) {- (PS'
emptyTrPtoQImpliesEq+      : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(P' : Process ∞ {lu} c)
                           (PS' : stable+ P)(tr : TrP+ {lu} [] (inj1 P') P)
                           → node P ≡ P'

emptyTrPtoQImpliesEq+ P P' PS' tr
    = emptyTrPtoQImpliesEq (node P) P' PS' (tnode tr)

```

--@BEGIN@bisimPPWithEmptyTr

mutual

```

bisimPPWithEmptyTr∞ : {lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P') (PS' : stable∞ P')
  (nonDivP : NonDivergent∞ P)
  (tr : TrP∞ {lu} []
    (inj1 (nonDivBecomeStable∞1 c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable∞1 c P nonDivP)
    (forcep P')
bisimPPWithEmptyTr∞ P P' PP' PS' nonDivP tr =
  bisimPPWithEmptyTr (forcep P) (forcep P')
    (forceB PP') PS' (forceND nonDivP) tr

```

```

bisimPPWithEmptyTr : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P') (PS' : stable P')
  (nonDivP : NonDivergent P)
  (tr : TrP {lu} [] (inj1
    (nonDivBecomeStable1 {lu} c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable1 {lu} c P nonDivP) P'
bisimPPWithEmptyTr {lu} {c} .(terminate x) (terminate x1)
  PP' PS' nonDivP (empty x) = PP'
bisimPPWithEmptyTr (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv nondivPI (inj1 x)) (tnode tr) =
  nonDivBecomesStableBisimProof∞ (PI P x)
  (nondivPI x) a (onlyIntChoice terequivP x)
bisimPPWithEmptyTr (node P) (terminate x)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv x1 (inj2 y)) (tnode tr) =
  eqterminater (termeqnode terequivP)
bisimPPWithEmptyTr (terminate P) (node P') PP' PS' nonDivP tr =
  PP'
bisimPPWithEmptyTr (node P) (node P') (eqnode bisimPP') PS'
  (nondiv x chemptyornot) (tnode tr) =
  bisimPPWithEmptyTr+ P P' bisimPP'
  PS' (nondiv x chemptyornot) tr

```

```

bisimPPWithEmptyTr+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P') (PS' : stable+ P')
  (nonDivP : NonDivergent+ P)
  (tr : TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable+1 c P nonDivP)
    (node P')
bisimPPWithEmptyTr+ {lu}{c} P P' PP' PS'
  (nondiv nondiv' (inj1 x1)) tr = PP'''
where
  P'~ : Process∞ ∞ {lu} c
  P'~ = bisimIP' PP' x1

```

```
trP'P'~ : P' → +*[ [] ] (forcep (P'~))
trP'P'~ = bisimltr PP' x1
```

```
P'≡P'~ : node P' ≡ forcep P'~ {∞}
P'≡P'~ = emptyTrPtoQImpliesEq+ P'
      (forcep P'~) PS' trP'P'~
```

```
P'~≡P' : forcep P'~ {∞} ≡ node P'
P'~≡P' rewrite P'≡P'~ = refl
```

```
P'~stable : stable (forcep P'~)
P'~stable rewrite P'~≡P' = PS'
```

```
PP'' : Bisimw (nonDivBecomeStable1 c
  (forcep (PI P x1)) (forceND (nondiv' x1)) )
  (forcep P'~)
PP'' = bisimPPWithEmptyTr (forcep (PI P x1))
  (forcep P'~ {∞})
  (forceB (bisimlnext PP' x1))
  P'~stable (forceND (nondiv' x1))
  (nonDivBecomeStable2 c
    (forcep (PI P x1)) (forceND (nondiv' x1)))
```

```
PP''' : Bisimw (nonDivBecomeStable∞1 c
  (PI P x1) (nondiv' x1)) (node P')
PP''' rewrite P'≡P'~ = PP''
```

```
bisimPPWithEmptyTr+ P P' PP' PS'
  (nondiv x (inj2 y)) empty = eqnode PP'
bisimPPWithEmptyTr+ P P' PP' PS'
  (nondiv x (inj2 y))
  (intc .[] .(inj1
    (node P)) x1 x2) = eqnode PP'
```

```
--@END
```

```
mutual
```

$\text{choicesetornotBism} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} + \infty \{lu\} c) (PP' : \text{Bisim} (cc' : \text{ChoiceSet } (I P) \sqcup \neg (\text{ChoiceSet } (I P))) \rightarrow \text{ChoiceSet } (I P') \sqcup \neg (\text{ChoiceSet } (I P')))$

$\text{choicesetornotBism } \{c\} P P' PP' (\text{inj}_1 \text{ ip}) = \text{inj}_1 (\text{bisim2l } PP' \text{ ip})$
 $\text{choicesetornotBism } \{c\} P P' PP' (\text{inj}_2 \text{ notip}) = \text{inj}_2 (\lambda \text{ ip}' \rightarrow \text{notip } (\text{bisim2lr } PP' \text{ ip}'))$

mutual

$\text{nondivLemBisims} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c) \rightarrow \text{Bisims} \{i\} P P' \rightarrow \text{NonDivergent} \{i\} P \rightarrow \text{NonDivergent} \{i\} P'$
 $\text{forceND } (\text{nondivLemBisims} P P' PP' nP) = \text{nondivLemBisims } (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP')$

$\text{nondivLemBisims} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c) \rightarrow \text{Bisims} \{i\} P P' \rightarrow \text{NonDivergent} \{i\} P \rightarrow \text{NonDivergent} \{i\} P'$
 $\text{nondivLemBisims} .(\text{terminate } a) .(\text{terminate } a) (\text{eqterminate } \{a\}) nP = \text{tt}$
 $\text{nondivLemBisims} .(\text{node } Q) .(\text{node } Q') (\text{eqnode } \{Q\} \{Q'\} QQ') nP = \text{nondivLemBisims} + Q$

$\text{nondivLemBisims} + : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} + \infty \{lu\} c) \rightarrow \text{Bisims} + \{i\} P P' \rightarrow \text{NonDivergent} + \{i\} P \rightarrow \text{NonDivergent} + \{i\} P'$
 $\text{nondivLemBisims} + P P' PP' (\text{nondiv } f p) =$
 $\text{nondiv } (\lambda i \rightarrow \text{nondivLemBisims} (\text{PI } P (\text{bisim2lr } PP' i)) (\text{PI } P' i) (\text{bisimlNext } PP' i) (f (\text{bisim2lr } PP' i)) (\text{choicesetornotBism } P P' PP' p))$

mutual

$\text{divLemBisims} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c) \rightarrow \text{Bisims} \{i\} P P' \rightarrow \text{DivergentProcess} i c P \rightarrow \text{DivergentProcess} i c P'$
 $\text{divLemBisims} P P' PP' nP .\text{forcediv} = \text{divLemBisims } (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP')$
 $\text{divLemBisims} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c) \rightarrow \text{Bisims} \{i\} P P' \rightarrow \text{DivergentProcess} i c P \rightarrow \text{DivergentProcess} i c P'$
 $\text{divLemBisims} .(\text{terminate } a) .(\text{terminate } a) (\text{eqterminate } \{a\}) nP = nP$

$\text{divLemBisims} \cdot (\text{node } P) \cdot (\text{node } P') (\text{eqnode } \{P\} \{P'\} PP') (\text{div } P \text{ div} P) = \text{div } P' (\text{divLemBisims} + P P')$

$\text{divLemBisims} + : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} + \infty \{lu\} c)$
 $\rightarrow \text{Bisims} + \{i\} P P'$
 $\rightarrow \text{DivergentProcess} + i c P \rightarrow \text{DivergentProcess} + i c P'$
 $\text{divLemBisims} + P P' PP' (\text{div} + \text{int } Q) = \text{div} + (\text{bisim2I } PP' \text{ int})$
 $(\text{divLemBisims} \infty (\text{PI } P \text{ int}) (\text{PI } P' (\text{bisim2I } PP' \text{ int})) (\text{bisimINext } PP' \text{ int}) Q)$

mutual

$\text{nondivLemBisims} \infty r : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \infty \{lu\} c)$
 $\rightarrow \text{Bisims} \infty \{i\} P P'$
 $\rightarrow \text{NonDivergent} \infty \{i\} P' \rightarrow \text{NonDivergent} \infty \{i\} P$
 $\text{forceND } (\text{nondivLemBisims} \infty r P P' PP' nP) = \text{nondivLemBisimsr } (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP' nP)$

$\text{nondivLemBisimsr} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c)$
 $\rightarrow \text{Bisims} \{i\} P P'$
 $\rightarrow \text{NonDivergent} \{i\} P' \rightarrow \text{NonDivergent} \{i\} P$
 $\text{nondivLemBisimsr} \cdot (\text{terminate } a) \cdot (\text{terminate } a) (\text{eqterminate } \{a\}) nP = \text{tt}$
 $\text{nondivLemBisimsr} \cdot (\text{node } Q) \cdot (\text{node } Q') (\text{eqnode } \{Q\} \{Q'\} QQ') nP = \text{nondivLemBisims} + r Q Q' QQ' nP$

$\text{nondivLemBisims} + r : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} + \infty \{lu\} c)$
 $\rightarrow \text{Bisims} + \{i\} P P'$
 $\rightarrow \text{NonDivergent} + \{i\} P' \rightarrow \text{NonDivergent} + \{i\} P$
 $\text{nondivLemBisims} + r P P' PP' (\text{nondiv } f p) =$
 $\text{nondiv } (\lambda i \rightarrow \text{nondivLemBisims} \infty r (\text{PI } P i) ((\text{PI } P' (\text{bisim2I } PP' i))) (\text{bisimINext } PP' i)$
 $(f (\text{bisim2I } PP' i))) (\text{swapChoiceSetssr } P P' PP' p))$

mutual

$\text{divLemBisims} \infty r : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \infty \{lu\} c)$
 $\rightarrow \text{Bisims} \infty \{i\} P P'$
 $\rightarrow \text{DivergentProcess} \infty i c P' \rightarrow \text{DivergentProcess} \infty i c P$
 $\text{divLemBisims} \infty r \{i\} P P' PP' nP \cdot \text{forcediv} = \text{divLemBisimsr } (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP' nP) (\text{forcediv})$
 $\text{divLemBisimsr} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process} \infty \{lu\} c)$
 $\rightarrow \text{Bisims} \{i\} P P'$

$$\rightarrow \text{DivergentProcess } i \ c \ P' \rightarrow \text{DivergentProcess } i \ c \ P$$

$$\begin{aligned} \text{divLemBisimsr} \ .(\text{terminate } a) \ .(\text{terminate } a) \ (\text{eqterminate } \{a\}) \ nP &= nP \\ \text{divLemBisimsr} \ .(\text{node } P') \ .(\text{node } P) \ (\text{eqnode } \{P'\} \ \{P\} \ PP') \ (\text{div } P \ \text{div} P) &= \text{div } P' \ (\text{divLemBisimsr} \ .(\text{node } P') \ .(\text{node } P) \ (\text{eqnode } \{P'\} \ \{P\} \ PP') \ (\text{div } P \ \text{div} P)) \end{aligned}$$

$$\begin{aligned} \text{divLemBisims+r} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process+ } \infty \ \{lu\} \ c) \\ \rightarrow \text{Bisims+ } \{i\} \ P \ P' \\ \rightarrow \text{DivergentProcess+ } i \ c \ P' \rightarrow \text{DivergentProcess+ } i \ c \ P \\ \text{divLemBisims+r} \ P \ P' \ PP' \ (\text{div+ } \text{int } Q) &= \text{div+ } ((\text{bisim2lr } PP' \ \text{int})) \\ &((\text{divLemBisims}\infty \ (\text{PI } P \ ((\text{bisim2lr } PP' \ \text{int})))) \ (\text{PI } P' \ \text{int})) \end{aligned}$$

mutual

$$\begin{aligned} \text{stabLemBisims}\infty : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process}\infty \ \{lu\} \ c) \\ \rightarrow \text{Bisims}\infty \ P \ P' \\ \rightarrow \text{stable}\infty \ P' \rightarrow \text{stable}\infty \ P \\ \text{stabLemBisims}\infty \ P \ P' \ PP' \ PS' &= \text{stabLemBisims} \ (\text{forcep } P) \ (\text{forcep } P') \ (\text{forceB } PP') \ PS' \\ \text{stabLemBisims} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process } \infty \ \{lu\} \ c) \\ \rightarrow \text{Bisims } P \ P' \\ \rightarrow \text{stable } P' \rightarrow \text{stable } P \\ \text{stabLemBisims} \ .(\text{terminate } a) \ .(\text{terminate } a) \ (\text{eqterminate } \{a\}) \ PS' &= PS' \\ \text{stabLemBisims} \ .(\text{node } P) \ .(\text{node } P') \ (\text{eqnode } \{P\} \ \{P'\} \ PP') \ PS' &= \text{stabLemBisims+ } P \ P' \ PP' \end{aligned}$$

$$\begin{aligned} \text{stabLemBisims+} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process+ } \infty \ \{lu\} \ c) \\ \rightarrow \text{Bisims+ } \{i\} \ P \ P' \\ \rightarrow \text{stable+ } P' \rightarrow \text{stable+ } P \\ \text{stabLemBisims+ } P \ P' \ PP' \ (PnoI \ , \ PNoTick) &= (\lambda \ \text{int} \rightarrow PnoI \ (\text{bisim2l } PP' \ \text{int})) \ , \ (\lambda \ t \rightarrow \perp) \end{aligned}$$

mutual

$$\begin{aligned} \text{divergentImpliesNotTermEquiv+} : \{lu : \text{LUniv}\} (c : \text{Choice}) \\ (P : \text{Process+ } \infty \ \{lu\} \ c) \\ (a : \text{ChoiceSet } c) \\ (\text{ter} P : \text{TerminateEquivalent+ } a \ P) \\ (\text{div} P : \text{DivergentProcess+ } \infty \ \{lu\} \ c \ P) \\ \rightarrow \perp \end{aligned}$$

```
divergentImpliesNotTermEquiv+ c P a terequivP (div+ int divP) =
  divergentImpliesNotTermEquiv∞ c (PI P int) a (onlyIntChoice terequivP int) divP
```

```
divergentImpliesNotTermEquiv∞ : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent∞ a P)
  (divP : DivergentProcess∞ ∞ {lu} c P)
  → ⊥
```

```
divergentImpliesNotTermEquiv∞ c P a terP divP = divergentImpliesNotTermEquiv c (forceP P) a terP
```

```
divergentImpliesNotTermEquiv : {lu : LUniv}(c : Choice)
  (P : Process ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent a P)
  (divP : DivergentProcess ∞ {lu} c P)
  → ⊥
```

```
divergentImpliesNotTermEquiv c .(node P) a (termenode terequivP) (div P divP) =
  divergentImpliesNotTermEquiv+ c P a terequivP
```

mutual

```
bisimImpliesDivergentPreserv∞ : {lu : LUniv}(c : Choice) (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ {∞} P P')
  (divP : DivergentProcess∞ ∞ {lu} c P)
  → DivergentProcess∞ ∞ {lu} c P'
```

```
forcediv (bisimImpliesDivergentPreserv∞ c P P' PP' divP) =
  bisimImpliesDivergentPreserv c (forceP P) (forceP P') (forceB PP')
  (forcediv divP)
```

```
bisimImpliesDivergentPreserv+ : {lu : LUniv}(c : Choice) (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (divP : DivergentProcess+ ∞ {lu} c P)
  → DivergentProcess+ ∞ {lu} c P'
```

```
bisimImpliesDivergentPreserv+ c P P' PP' divP = bisimdiv PP' divP
```

```
--@BEGIN@bisimImpliesDivergentPreserv
```

```

bisimImpliesDivergentPreserv :      {lu : LUniv}{c : Choice)
                                     (P P' : Process ∞ {lu} c)
                                     (PP' : Bisimw {∞} P P')
                                     (divP : DivergentProcess ∞ {lu} c P)
                                     → DivergentProcess ∞ {lu} c P'
bisimImpliesDivergentPreserv c .(terminate _) P' (eqterminate x) ()
bisimImpliesDivergentPreserv c .(node P) .(terminate a) (eqterminater {a}
  {(node P)} (termeqnode terequivP)) (div P divP)
  = ⊥-elim (divergentImpliesNotTermEquiv+ c P a terequivP divP)
bisimImpliesDivergentPreserv c .(node P) .(node P') (eqnode {P} {P'} PP')
  (div P divP)
  = div P' (bisimImpliesDivergentPreserv+ c P P' PP' divP)

```

--@END

mutual

```

bisimStableImpliesNotDivergent∞' : {lu : LUniv}{c : Choice) (P P' : Process∞ ∞ {lu} c)
                                     (PP' : Bisimw∞ P P')
                                     (PS' : stable∞ P')
                                     → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞' c P P' PP' PS') =
  bisimStableImpliesNotDivergent' c (forceP P)
  (forceP P') (forceB PP') PS'

bisimStableImpliesNotDivergent' : {lu : LUniv}{c : Choice) (P P' : Process ∞ {lu} c)
                                   (PP' : Bisimw P P')
                                   (PS' : stable P')
                                   → NonDivergent P
bisimStableImpliesNotDivergent' c (terminate x) P' PP' PS' = tt
bisimStableImpliesNotDivergent' c (node P) .(terminate a)
  (eqterminater {a} terequiv) PS' =
  TerImpliesNotDivergentaux c (node P) a terequiv
bisimStableImpliesNotDivergent' c (node P) .(node P')
  (eqnode {P} {P'} PP') PS' =
  bisimStableImpliesNotDivergent+' c P P' PP' PS'

bisimStableImpliesNotDivergent+' : {lu : LUniv}{c : Choice) (P P' : Process+ ∞ {lu} c)
                                   (PP' : Bisimw+ P P') (PS' : stable+ P')

```

\rightarrow NonDivergent+ P
 bisimStableImpliesNotDivergent+' c P P' PP' PS' =
 nondiv+r PP' (nondiv $(\lambda i \rightarrow \perp\text{-elim (stabToNoInternal+ } P' PS' i))$ (inj₂ (s

lemBisimDRefusalAux : { lu : LUniv}{ c : Choice}
 (x : ChoiceSet c)
 (P : Process+ ∞ { lu } c)
 ($hasTauorTickNoTau$: ChoiceSet ($\mid P$) \sqcup (\neg (ChoiceSet ($\mid P$)) \times ChoiceSet ($\mid P$)) $\rightarrow \perp$)
 (PS : ChoiceSet ($\mid P$) $\rightarrow \perp$)
 \rightarrow ChoiceSet (\top P)
 lemBisimDRefusalAux c x P (inj₁ x_1) PS = $\perp\text{-elim (PS } x_1)$
 lemBisimDRefusalAux c x P (inj₂ ($-$, x_1)) PS = x_1

mutual

bisimDRefusal+ : { lu : LUniv}{ c : Choice} (P : Process+ ∞ { lu } c) (P' : Process+ ∞ { lu } c)
 (PS : stable+ P)
 (PP' : Bisimw+ { ∞ } P P')
 (X : Label $lu \rightarrow$ Bool)
 ($isRoscoe$: Bool)
 (ref : DRefusal+ P $isRoscoe$ X)
 \rightarrow DRefusal+ P' $isRoscoe$ X
 bisimDRefusal+ { c } P P' PS PP' X $isRoscoe$ (drefusal noextChInX noTerm) =
 drefusal (bisimDRefusal+NoExtChInX P P' PS PP' X noextChInX)
 (bisimDRefusalNoTicksIfsRoscoe P P' PS PP' $isRoscoe$ noTerm)

bisimDRefusalNoTicksIfsRoscoe : { lu : LUniv}{ c : Choice} (P : Process+ ∞ { lu } c)
 (P' : Process+ ∞ { lu } c)
 (PS : stable+ P)
 (PP' : Bisimw+ { ∞ } P P')
 ($isRoscoe$: Bool)
 (ref : NoTicksIfsRoscoe P $isRoscoe$)
 \rightarrow NoTicksIfsRoscoe P' $isRoscoe$

bisimDRefusalNoTicksIfsRoscoe { lu } { c } P P' PS PP' $isRoscoe$ ref $tickIsIncl$ x =
 ref $tickIsIncl$ (lem c P (\top P' x) path PS)
 where
 path : TrP+ { lu } [] (inj₂ (\top P' x)) P

path = bisimTtrr PP' x

lem : {lu : LUniv} (c : Choice) (P : Process+ ∞ {lu} c) (x : ChoiceSet c)
 (tr : TrP+ {lu} [] (inj₂ x) P)
 (PS : stable+ P) → ChoiceSet (T P)
 lem c P x (intc .[] .(inj₂ x) x' x₂) PS = ⊥-elim (stabToNoInternal+ P PS x')
 lem c P .(PT P x₁) (terc x₁) PS = x₁

bisimDRefusal+NoExtChInX : {lu : LUniv} {c : Choice} (P : Process+ ∞ {lu} c)
 (P' : Process+ ∞ {lu} c)
 (PS : stable+ P)
 (PP' : Bisimw+ {∞} P P')
 (X : Label lu → Bool)
 (ref : NoExtChInX P X)
 → NoExtChInX P' X

bisimDRefusal+NoExtChInX {lu} {c} P P' PS PP' X ref e x = lem2 c P Q (Lab P' e) tr
 where
 Q : Process ∞ {lu} c
 Q = forcep (PP' .bisimEP'r e)
 tr : TrP+ {lu} (Lab P' e :: []) (inj₁ Q) P
 tr = PP' .bisimEtrr e

lem2 : {lu : LUniv} (c : Choice) (P : Process+ ∞ {lu} c) (Q : Process ∞ {lu} c)
 (l : Label lu) (tr : TrP+ {lu} (l :: []) (inj₁ Q) P)
 (PS : stable+ P) (X : Label lu → Bool) (ref : NoExtChInX P X) (labX : True (X l))
 lem2 c P Q .(Lab P x) (extc .[] .(inj₁ Q) x x₁) PS X ref labX = ref x labX
 lem2 c P Q l (intc .(l :: []) .(inj₁ Q) x x₁) PS X ref labX = stabToNoInternal+ P PS x

bisimDRefusal : {lu : LUniv} {c : Choice} (P : Process ∞ {lu} c) (P' : Process ∞ {lu} c)
 (PS : stable P)
 (PP' : Bisimw {∞} P P')
 (X : Label lu → Bool)
 (isRoscoe : Bool)
 (ref : DRefusal P isRoscoe X)
 → DRefusal P' isRoscoe X

bisimDRefusal (terminate x) (terminate x₁) PS PP' X isRoscoe ref tickIncl = ref tickIncl

```

bisimDRefusal (terminate x) (node Q) PS      (eqterminate (termeqnode terequivP)) X isRoscoe
                                     drefusal (λ e → ⊥-elim (noExtChoice terequivP e))(λ t → ⊥-elim (
bisimDRefusal {lu}{c} (node P) (terminate x) PS
    (eqterminater (termeqnode terequivP)) X isRoscoe (drefusal noextChInX noTerm) tickInc
    = noTerm tickInc (lemBisimDRefusalAux c x P (hasTauOrTickNoTau terequivP) (stabToN
bisimDRefusal (node P) (node P') PS      (eqnode bisimQQ') X isRoscoe ref =
                                     bisimDRefusal+ P P' PS bisimQQ'

```

mutual

```

lemmaxxx1 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (Q : Process ∞ {lu} c)
    → (divQ : DivergentProcess ∞ {lu} c Q)
    → BisimForNextP      result (inj1 Q)
    → Process ∞ {lu} c
lemmaxxx1 (inj1 Q') Q divQ BisimResultQ = Q'
lemmaxxx1 (inj2 y) Q divQ BisimResultQ = ⊥-elim (lemmaDivNotTermequiv Q divQ y BisimResultQ)

lemmaxxx2 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
    → (divQ : DivergentProcess ∞ {lu} c Q)
    → (bisimForNext : BisimForNextP      result (inj1 Q))
    → (trp : TrP {lu} l result P)
    → TrP {lu} l (inj1 (lemmaxxx1 result Q divQ bisimForNext)) P
lemmaxxx2 (inj1 x) l P Q divQ bisimForNext trp = trp
lemmaxxx2 (inj2 y) l P Q divQ bisimForNext trp = ⊥-elim (lemmaDivNotTermequiv Q divQ y bisimForNext)

lemmaxxx2+ : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
    → (divQ : DivergentProcess ∞ {lu} c Q)
    → (bisimForNext : BisimForNextP      result (inj1 Q))
    → (trp+ : TrP+ {lu} l result P)
    → TrP+ {lu} l (inj1 (lemmaxxx1 result Q divQ bisimForNext)) P
lemmaxxx2+ (inj1 x) l P Q divQ bisimForNext trp+ = trp+
lemmaxxx2+ (inj2 y) l P Q divQ bisimForNext trp+ = ⊥-elim (lemmaDivNotTermequiv Q divQ y bisimForNext)

lemmaxxx3 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
    → (l : List (Label lu))(Q : Process ∞ {lu} c)
    → (divQ : DivergentProcess ∞ {lu} c Q)

```

```

→ (bisimForNext : BisimForNextP      result (inj1 Q))
→ Bisimw (lemmaxxx1 result Q divQ bisimForNext) Q
lemmaxxx3 (inj1 Q') l Q divQ bisimForNext = bisimForNext
lemmaxxx3 (inj2 y) l Q divQ bisimForNext = ⊥-elim (lemmaDivNotTermequiv Q divQ y bisimForNext)

```

mutual

```

lemmayyy1 : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true      X
→ BisimForNextP      result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1 (inj1 Q') Q      stab X x x1 = Q'
lemmayyy1 (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X      dref stab y termequivQ)

```

```

lemmayyy1+ : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true      X
→ BisimForNextP      result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1+ (inj1 Q') Q      stab X x x1 = Q'
lemmayyy1+ (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy2 : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
→ (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true      X)
→ (bisim : BisimForNextP      result (inj1 Q))
→ TrP {lu} l result P
→ TrP {lu} l (inj1 (lemmayyy1 result Q      stab X dref bisim)) P
lemmayyy2 (inj1 Q') l P Q      stab X dref bisim tr = tr

```

```
lemmayyy2 (inj2 y) l P Q      stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)
```

```
lemmayyy2+ : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab+ : stable Q)
  → (X : Label lu → Bool)
  → (dref : DRefusal {lu}{c} Q true      X)
  → (bisim : BisimForNextP      result (inj1 Q))
  → TrP+ {lu} l result P
  → TrP+ {lu} l (inj1 (lemmayyy1 result Q stab+ X dref bisim)) P
lemmayyy2+ (inj1 Q') l P Q      stab X dref bisim tr = tr
lemmayyy2+ (inj2 y) l P Q      stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)
```

```
lemmayyy3 : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (X : Label lu → Bool)
  → (dref : DRefusal {lu}{c} Q true      X)
  → (bisim : BisimForNextP      result (inj1 Q))
  → Bisimw (lemmayyy1 result Q      stab X dref bisim) Q
lemmayyy3 (inj1 Q') Q      stab X dref bisim = bisim
lemmayyy3 (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)
```

```
lemEqList : {A : Set}(l : List A) → l ++ [] ≡ l
lemEqList [] = refl
lemEqList (x :: l) = cong (λ l' → x :: l') (lemEqList l)
```

```
stableNotTerminateEquivaux : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (hasTauOrTickNoTau : ChoiceSet (| P) ⊕
```

$$\begin{aligned}
 & (\neg (\text{ChoiceSet } (\text{I } P)) \times \text{ChoiceSet } (\text{T } P))) \\
 & (\text{stab} : \text{stable+ } P) \\
 & (\text{notick} : \text{noTickIfRoscoe+ true } P) \\
 & \rightarrow \perp
 \end{aligned}$$

$$\begin{aligned}
 \text{stableNotTerminateEquiv} & P x (\text{inj}_1 \text{ int}) (\text{stabsch} \text{ ,, } x_2) \text{ notick} = \text{stabsch int} \\
 \text{stableNotTerminateEquiv} & P x (\text{inj}_2 (\text{noint} \text{ ,, tick})) \text{ stab notick} = \text{notick tick}
 \end{aligned}$$

$$\begin{aligned}
 \text{stableNotTerminateEquiv} & : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\
 & (x : \text{ChoiceSet } c) \\
 & (\text{terequiv} : \text{TerminateEquivalent } x P) \\
 & (\text{stab} : \text{stable } P) \\
 & \rightarrow \perp
 \end{aligned}$$

$$\begin{aligned}
 \text{stableNotTerminateEquiv} & (\text{terminate } x) x_1 \text{ terequiv stab} = \text{stab} _ \\
 \text{stableNotTerminateEquiv} & (\text{node } P) x (\text{termeqnode terequiv } P) (\text{stabSch} \text{ ,, notick}) \\
 & = \text{stableNotTerminateEquiv} P x (\text{hasTauOrTickNoTau terequiv } P) (\text{stabSch} \text{ ,, notick}) \text{ no}
 \end{aligned}$$

$$\begin{aligned}
 \text{noIntNoTerImpliesNoTermTrace} & : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+} \infty \{lu\} c) \\
 & (x : \text{ChoiceSet } c) \\
 & (\text{tr} : \text{TrP+ } [] (\text{inj}_2 x) P) \\
 & (\text{noInt} : \neg (\text{ChoiceSet } (\text{I } P))) \\
 & (\text{noTer} : \neg (\text{ChoiceSet } (\text{T } P))) \\
 & \rightarrow \perp
 \end{aligned}$$

$$\begin{aligned}
 \text{noIntNoTerImpliesNoTermTrace} & P x (\text{intc} \text{ .} [] (\text{inj}_2 x) \text{ int } x_2) \text{ noInt noTer} = \text{noInt int} \\
 \text{noIntNoTerImpliesNoTermTrace} & P \text{ .} (\text{PT } P \text{ ter}') (\text{terc } \text{ ter}') \text{ noInt noTer} = \text{noTer ter}'
 \end{aligned}$$

mutual

$$\begin{aligned}
 \text{bisimwStableToNoTick} & : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) \\
 & (P' : \text{Process} \infty \{lu\} c) \\
 & (PP' : \text{Bisimw } \{\infty\} P P') \\
 & (\text{stabP}' : \text{stable } P') \\
 & (\text{stabP} : \text{stableSch } P) \\
 & \rightarrow \text{noTickIfRoscoe true } P
 \end{aligned}$$

$$\begin{aligned}
 \text{bisimwStableToNoTick} & (\text{terminate } x) P' (\text{eqterminate terequiv}) \text{ stabP}' \text{ stabP} = \\
 & \text{stableNotTerminateEquiv } P'
 \end{aligned}$$

$$\begin{aligned}
 \text{bisimwStableToNoTick} & (\text{terminate } x) \text{ .} (\text{terminate} _) (\text{eqterminater terequiv}) \text{ stabP}' \text{ stabP} = s \\
 \text{bisimwStableToNoTick} & (\text{node } P) (\text{terminate } x) PP' \text{ stabP}' \text{ stabP} t = \text{stabP}' _
 \end{aligned}$$

$$\begin{aligned}
 \text{bisimwStableToNoTick} & (\text{node } P) (\text{node } P') (\text{eqnode PP}') (\text{noint} \text{ ,, noterP}') \text{ noterP ter}' \\
 & = \text{noIntNoTerImpliesNoTermTrace } P' (\text{PT } P \text{ ter}') (\text{bisimTtr PP}' \text{ ter}') \text{ noint noterP}'
 \end{aligned}$$

mutual

--@BEGIN@bisimRefusalros

```

bisimRefusalros : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P') (l : List (Label lu))
  (X : Label lu → Bool)
  (fail : failure P' l true      X)
  → failure P l true X

```

```

bisimRefusalros {lu}{c} P P' PP' l X
  (stableFail (stableFp Q' tr' stab' drefuse'))
  = (stableFail (stableFp Qhat trhat2
    (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat
    stabNoTick )
    drefusehat))

```

where

```

Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
Qcom  = bisimTraceTrP1 P P' PP' l (inj1 Q') tr'

```

```

trcom : TrP {lu} l (bisimTraceTrP1 P P' PP' l
  (inj1 Q') tr') P
trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') tr'

```

```

QQ'com : BisimForNextP (bisimTraceTrP1 P P' PP' l
  (inj1 Q') tr') (inj1 Q')
QQ'com = bisimTraceTrP3 P P' PP' l (inj1 Q') tr'

```

```

Q : Process ∞ {lu} c
Q = lemmayyy1 Qcom Q'      stab' X drefuse' QQ'com

```

```

tr : TrP {lu} l (inj1 Q) P
tr = lemmayyy2 Qcom l P Q'      stab' X drefuse' QQ'com trcom

```

```

QQ' : Bisimw Q Q'
QQ' = lemmayyy3 Qcom Q'      stab' X drefuse' QQ'com

```

```

Qhat : Process ∞ {lu} c
Qhat  = nonDivBecomeStable1 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
      (stableImpliesNonDiv Q' stab'))

```

```

trhat : TrP {lu} [] (inj1 Qhat) Q
trhat   = nonDivBecomeStable2 c Q
          (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
           (stableImpliesNonDiv Q' stab'))

```

```

QhatQ' :      Bisimw Qhat Q'
QhatQ' =      bisimPPWithEmptyTr Q Q' QQ' stab'
              (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
               (stableImpliesNonDiv Q' stab')) trhat

```

```

stabSchQhat : stableSch Qhat
stabSchQhat   = nonDivBecomeStable3 c Q
               (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
                (stableImpliesNonDiv Q' stab'))

```

```

stabNoTick : noTickIfRoscoe true Qhat
stabNoTick   = bisimwStableToNoTick Qhat Q' QhatQ' stab' stabSchQhat

```

```

trhat1 : TrP {lu} (l ++ []) (inj1 Qhat) P
trhat1 = trPAppendTrw c P Q l [] (inj1 Qhat) tr trhat

```

```

eq1 :      (l ++ [] ) ≡ l
eq1 = lemEqList l

```

```

trhat2 : TrP {lu} l (inj1 Qhat) P
trhat2 = subst (λ l' → TrP {lu} l' (inj1 Qhat) P) eq1 trhat1

```

```

drefusehat :      DRefusal Qhat true X
drefusehat =      bisimDRefusal Q' Qhat stab'
                  (BismwSym Qhat Q' QhatQ') X true drefuse'

```

```

bisimRefusalros {lu}{c} P P' PP' l X
                  (divergentFailure (trdiv Q' trp' divq'))
                  = (divergentFailure (trdiv Q tr divp))

```

where

```

Qcom :      Process ∞ {lu} c ⊔ ChoiceSet c
Qcom   = bisimTraceTrP1 P P' PP' l (inj1 Q') trp'

```

```

trcom : TrP {lu} l (bisimTraceTrP1 P P' PP' l (inj1 Q') trp') P
trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') trp'

```

```

QQ'com : BisimForNextP
      (bisimTraceTrP1 P P' PP' l (inj1 Q') trp') (inj1 Q')
QQ'com = bisimTraceTrP3 P P' PP' l (inj1 Q') trp'

Q : Process ∞ {lu} c
Q = lemmaxx1 Qcom Q' divq' QQ'com

tr : TrP {lu} l (inj1 Q) P
tr = lemmaxx2 Qcom l P Q' divq' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmaxx3 Qcom l Q' divq' QQ'com

Q'Q : Bisimw Q' Q
Q'Q = BismwSym Q Q' QQ'

divp : DivergentProcess ∞ {lu} c Q
divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divq'

--@END

--@BEGIN@bisimRefusalrosplus

mutual

bisimRefusalros+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
      (P' : Process+ ∞ {lu} c)
      (PP' : Bisimw+ {∞} P P')
      (l : List (Label lu))
      (X : Label lu → Bool)
      (fail : failure+ P' l true X)
      → failure+ P l true X

bisimRefusalros+ {lu}{c} P P' PP' l X
      (stableFail (stableFp Q' tr' stab' drefuse'))
      = (stableFail (stableFp Qhat trhat2
      (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat
      stabNoTick)
      drefusehat))

where
  Qcom : Process ∞ {lu} c ⊔ ChoiceSet c

```

$$Qcom = bisimTraceTrP_1 + P P' PP' l (inj_1 Q') tr'$$

$$\begin{aligned} trcom &: TrP + l (bisimTraceTrP_1 + P P' PP' l (inj_1 Q') tr') P \\ trcom &= bisimTraceTrP_2 + P P' PP' l (inj_1 Q') tr' \end{aligned}$$

$$\begin{aligned} QQ'com &: BisimForNextP \\ &\quad (bisimTraceTrP_1 + P P' PP' l (inj_1 Q') tr') (inj_1 Q') \\ QQ'com &= bisimTraceTrP_3 + P P' PP' l (inj_1 Q') tr' \end{aligned}$$

$$\begin{aligned} Q &: Process \infty \{lu\} c \\ Q &= lemmayyy_1 Qcom Q' \quad stab' X drefuse' QQ'com \end{aligned}$$

$$\begin{aligned} tr &: TrP + \{lu\} l (inj_1 Q) P \\ tr &= lemmayyy_2 + Qcom l P Q' stab' X drefuse' QQ'com trcom \end{aligned}$$

$$\begin{aligned} QQ' &: Bisimw Q Q' \\ QQ' &= lemmayyy_3 Qcom Q' \quad stab' X drefuse' QQ'com \end{aligned}$$

$$\begin{aligned} Qhat &: Process \infty \{lu\} c \\ Qhat &= nonDivBecomeStable_1 c Q \\ &\quad (bisimStableImpliesNotDivergent c Q Q' QQ' stab' \\ &\quad \quad (stableImpliesNonDiv Q' stab')) \end{aligned}$$

$$\begin{aligned} trhat &: TrP \{lu\} [] (inj_1 Qhat) Q \\ trhat &= nonDivBecomeStable_2 c Q \\ &\quad (bisimStableImpliesNotDivergent c Q Q' QQ' stab' \\ &\quad \quad (stableImpliesNonDiv Q' stab')) \end{aligned}$$

$$\begin{aligned} QhatQ' &: Bisimw Qhat Q' \\ QhatQ' &= bisimPPWithEmptyTr Q Q' QQ' stab' \\ &\quad (bisimStableImpliesNotDivergent c Q Q' QQ' stab' \\ &\quad \quad (stableImpliesNonDiv Q' stab')) trhat \end{aligned}$$

$$\begin{aligned} stabSchQhat &: stableSch Qhat \\ stabSchQhat &= nonDivBecomeStable_3 c Q \\ &\quad (bisimStableImpliesNotDivergent c Q Q' QQ' stab' \\ &\quad \quad (stableImpliesNonDiv Q' stab')) \end{aligned}$$

$$\begin{aligned} stabNoTick &: noTickIfRoscoe true Qhat \\ stabNoTick &= bisimwStableToNoTick Qhat Q' QhatQ' stab' stabSchQhat \end{aligned}$$



$\text{trhat}_1 : \text{TrP}+ \{lu\} (l ++ []) (\text{inj}_1 \text{ Qhat}) P$
 $\text{trhat}_1 = \text{trPAppendTrw}+ c P Q l [] (\text{inj}_1 \text{ Qhat}) \text{ tr trhat}$

$\text{eq}l : (l ++ []) \equiv l$
 $\text{eq}l = \text{lemEqList } l$

$\text{trhat}_2 : \text{TrP}+ \{lu\} l (\text{inj}_1 \text{ Qhat}) P$
 $\text{trhat}_2 = \text{subst } (\lambda l' \rightarrow \text{TrP}+ \{lu\} l' (\text{inj}_1 \text{ Qhat}) P) \text{ eq}l \text{ trhat}_1$

$\text{drefusehat} : \text{DRefusal } \text{Qhat } \text{true } X$
 $\text{drefusehat} = \text{bisimDRefusal } Q' \text{ Qhat } \text{stab}'$
 $(\text{BismwSym } \text{Qhat } Q' \text{ QhatQ}') X \text{true } \text{drefuse}'$

$\text{bisimRefusalros}+ \{lu\}\{c\} P P' PP' l X$
 $(\text{divergentFailure } (\text{trdiv } Q' \text{ trp}' \text{ divq}'))$
 $= (\text{divergentFailure } (\text{trdiv } Q \text{ tr divp}))$

where

$\text{Qcom} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1+ \{lu\} P P' PP' l (\text{inj}_1 Q') \text{ trp}'$

$\text{trcom} : \text{TrP}+ \{lu\} l (\text{bisimTraceTrP}_1+ P P' PP' l (\text{inj}_1 Q') \text{ trp}') P$
 $\text{trcom} = \text{bisimTraceTrP}_2+ P P' PP' l (\text{inj}_1 Q') \text{ trp}'$

$\text{QQ'com} : \text{BisimForNextP}$
 $(\text{bisimTraceTrP}_1+ P P' PP' l (\text{inj}_1 Q') \text{ trp}') (\text{inj}_1 Q')$
 $\text{QQ'com} = \text{bisimTraceTrP}_3+ P P' PP' l (\text{inj}_1 Q') \text{ trp}'$

$Q : \text{Process } \infty \{lu\} c$
 $Q = \text{lemmaxxx}_1 \text{ Qcom } Q' \text{ divq}' \text{ QQ'com}$

$\text{tr} : \text{TrP}+ \{lu\} l (\text{inj}_1 Q) P$
 $\text{tr} = \text{lemmaxxx}_2+ \text{Qcom } l P Q' \text{ divq}' \text{ QQ'com } \text{trcom}$

$\text{QQ}' : \text{Bisimw } Q Q'$
 $\text{QQ}' = \text{lemmaxxx}_3 \text{ Qcom } l Q' \text{ divq}' \text{ QQ'com}$

$Q'Q : \text{Bisimw } Q' Q$
 $Q'Q = \text{BismwSym } Q Q' \text{ QQ}'$



```

divp : DivergentProcess  $\infty$  {lu} c Q
divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divq'

--@END

--@BEGIN@bisimImFdiTwo

bisimImFDI2 : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  → P  $\sqsubseteq$ fdi2ros P'
bisimImFDI2 {lu}{c} P P' PP' = bisimRefusalros P P' PP'

bisimImFDI2r : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  → P'  $\sqsubseteq$ fdi2ros P
bisimImFDI2r {lu}{c} P P' PP' = bisimImFDI2 P' P (BismwSym P P' PP')

--@END

bisimImFDI2+ : {lu : LUniv}{c : Choice} (P : Process+  $\infty$  {lu} c)
  (P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
  → P  $\sqsubseteq$ fdi2ros+ P'
bisimImFDI2+ {lu}{c} P P' PP' = bisimRefusalros+ P P' PP'

bisimImFDI2r+ : {lu : LUniv}{c : Choice} (P : Process+  $\infty$  {lu} c)
  (P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
  → P'  $\sqsubseteq$ fdi2ros+ P
bisimImFDI2r+ {lu}{c} P P' PP' = bisimImFDI2+ P' P (BismwSym+ P P' PP')

```

A.8 bisimilarityProofsWithSchneiderStable2.agda

```
--@PREFIX@bisimilarityProofs
```

```
module bisimilarityProofsWithSchneiderStable2
```

```
where
```

```

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool          renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData

```

mutual

```

nondivImpliesIPorNotIP : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
                        (nondiv : NonDivergent+ P)
                        → ChoiceSet (I P) ⊕ ¬ (ChoiceSet (I P))
nondivImpliesIPorNotIP {c} P
  (nondiv - chemptyornot) = chemptyornot

```

mutual

```

swapChoiceSets : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
                (PP' : Bisimw+ {∞} P P')
                (nondiv : NonDivergent+ P)
                → ChoiceSet (I P') ⊕ ¬ (ChoiceSet (I P'))
swapChoiceSets {c} P P' PP' nond
  = nondivImpliesIPorNotIP P' (nondiv+ PP' nond)

```

mutual

$\text{stableImpliesNonDiv}\infty : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process}\infty \infty \{lu\} c)$
 $(PS : \text{stable}\infty P) \rightarrow \text{NonDivergent}\infty P$

$\text{forceND } (\text{stableImpliesNonDiv}\infty P PS) = \text{stableImpliesNonDiv } (\text{forcep } P) PS$

$\text{stableImpliesNonDiv} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process} \infty \{lu\} c)$
 $(PS : \text{stable } P) \rightarrow \text{NonDivergent } P$

$\text{stableImpliesNonDiv } (\text{terminate } x) PS = \text{tt}$

$\text{stableImpliesNonDiv } (\text{node } x) PS = \text{stableImpliesNonDiv+ } x PS$

$\text{stableImpliesNonDiv+} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process+} \infty \{lu\} c)$
 $(PS : \text{stable+ } P) \rightarrow \text{NonDivergent+ } P$

$\text{stableImpliesNonDiv+ } P PS = \text{nondiv}$

$(\text{stableImpliesNonDiv+aux } P PS) (\text{inj}_2 (\text{stabToNoInternal+ } P PS))$

$\text{stableImpliesNonDiv+aux} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}(P : \text{Process+} \infty \{lu\} c)$
 $(PS : \text{stable+ } P)(i : \text{ChoiceSet } (\text{I } P))$
 $\rightarrow \text{NonDivergent}\infty (\text{PI } P i)$

$\text{stableImpliesNonDiv+aux } P PS i = \perp\text{-elim } (\text{stabToNoInternal+ } P PS i)$

mutual

$\text{TerImpliesNotDivergentaux+} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process+} \infty \{lu\} c)$
 $(a : \text{ChoiceSet } c)$

$(\text{terequiv} : \text{TerminateEquivalent+ } a P)$

$\rightarrow \text{NonDivergent+ } P$

$\text{TerImpliesNotDivergentaux+ } c P a$

$\text{terequiv} = \text{nondiv}$

$(\lambda i \rightarrow \text{TerImpliesNotDivergentaux}\infty c$

$(\text{PI } P i) a (\text{onlyIntChoice } \text{terequiv } i))$

$(\text{hasTauOrNotTau } \text{terequiv}))$

$\text{TerImpliesNotDivergentaux} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process} \infty \{lu\} c)$
 $(a : \text{ChoiceSet } c)$

$(\text{terequiv} : \text{TerminateEquivalent } a P)$

$\rightarrow \text{NonDivergent } P$

$\text{TerImpliesNotDivergentaux } c (\text{terminate } x) a \text{terequiv} = \text{tt}$

```
TerImpliesNotDivergentaux c (node x) a (termeqnode terequivP)
  = TerImpliesNotDivergentaux+ c x a terequivP
```

```
TerImpliesNotDivergentaux∞ : {lu : LUniv}(c : Choice)(P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent a (forceP P))
  → NonDivergent∞ P
```

```
forceND (TerImpliesNotDivergentaux∞ c P a terequiv)
  = TerImpliesNotDivergentaux c (forceP P) a terequiv
```

```
--@BEGIN@bisimStableImpliesNotDivergent
```

```
mutual
```

```
bisimStableImpliesNotDivergent∞ : {lu : LUniv}(c : Choice)
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (PS' : stable∞ P')
  (nonDivP' : NonDivergent∞ P')
  → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞ c P P' PP' PS' nonDivP')
  = bisimStableImpliesNotDivergent c (forceP P)
    (forceP P')
    (forceB PP')
    PS' (forceND nonDivP')
```

```
bisimStableImpliesNotDivergent : {lu : LUniv}(c : Choice)
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (PS' : stable P')
  (nonDivP' : NonDivergent P')
  → NonDivergent P
bisimStableImpliesNotDivergent c (terminate x) P' PP' PS' nonDivP' = tt
bisimStableImpliesNotDivergent c (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' nonDivP' =
  TerImpliesNotDivergentaux c (node P)
    a ((termeqnode terequivP))
```

```
bisimStableImpliesNotDivergent c (node P) (node P') (eqnode PP') PS'
```

$$nonDivP' = \text{bisimStableImpliesNotDivergent+} \\ c \ P \ P' \ PP' \ PS' \ nonDivP'$$

```

bisimStableImpliesNotDivergent+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (PS' : stable+ P')
  (nonDivP' : NonDivergent+ P')
  → NonDivergent+ P
bisimStableImpliesNotDivergent+ c P P' PP' PS' nonDivP'
  = nondiv+r PP' nonDivP'
--@END

```

```

--@BEGIN@nonDivBecomeStable

```

```

mutual

```

```

nonDivBecomeStable∞1 : {lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → Process ∞ {lu} c
nonDivBecomeStable∞1 c P nonDivP = nonDivBecomeStable1
  c (forcep P) (forceND nonDivP)

```

```

nonDivBecomeStable∞2 : {lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → TrP∞ {lu} [(inj1
    (nonDivBecomeStable∞1 c P nonDivP))] P
nonDivBecomeStable∞2 c P nonDivP = nonDivBecomeStable2
  c (forcep P) (forceND nonDivP)

```

```

nonDivBecomeStable∞3Sch : {lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → stableSch (nonDivBecomeStable∞1
    c P nonDivP)
nonDivBecomeStable∞3Sch c P nonDivP = nonDivBecomeStable3Sch
  c (forcep P) (forceND nonDivP)

```

```

nonDivBecomeStable+1 : {lu : LUniv}(c : Choice)
  (P : Process+ ∞ {lu} c)
  (nonDivP : NonDivergent+ P)
  → Process ∞ {lu} c
nonDivBecomeStable+1 c P (nondiv x (inj1 int)) =
  nonDivBecomeStable∞1
    c (PI P int) (x int)
nonDivBecomeStable+1 c P (nondiv x (inj2 stab)) = node P

nonDivBecomeStable+2 : {lu : LUniv}(c : Choice)
  (P : Process+ ∞ {lu} c)
  (nonDivP : NonDivergent+ P)
  → TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P
nonDivBecomeStable+2 c P (nondiv x (inj1 int)) = intc [] (inj1
  (nonDivBecomeStable+1 c P
    (nondiv x (inj1 int)))) int
  (nonDivBecomeStable∞2 c
    (PI P int) (x int))
nonDivBecomeStable+2 c P (nondiv x (inj2 stab)) = empty

nonDivBecomeStable+3Sch : {lu : LUniv}(c : Choice)
  (P : Process+ ∞ {lu} c)
  (nonDivP : NonDivergent+ P)
  → stableSch
    (nonDivBecomeStable+1 c P nonDivP)
nonDivBecomeStable+3Sch c P (nondiv x (inj1 int)) =
  nonDivBecomeStable∞3Sch c
    (PI P int) (x int)
nonDivBecomeStable+3Sch c P (nondiv x (inj2 stab)) = stab

nonDivBecomeStable1 : {lu : LUniv}(c : Choice)
  (P : Process ∞ {lu} c)
  (nonDivP : NonDivergent P)
  → Process ∞ {lu} c
nonDivBecomeStable1 c (terminate x) nonDivP = terminate x
nonDivBecomeStable1 c (node x) nonDivP =
  nonDivBecomeStable+1 c x nonDivP

nonDivBecomeStable2 : {lu : LUniv}(c : Choice)

```

```

(P : Process ∞ {lu} c)
(nonDivP : NonDivergent P)
→ TrP {lu} [] (inj1
  (nonDivBecomeStable1 c P nonDivP)) P
nonDivBecomeStable2 c (terminate x) nonDivP = empty x
nonDivBecomeStable2 c (node x) nonDivP
  = tnode (nonDivBecomeStable+2 c x nonDivP)

```

```

nonDivBecomeStable3Sch : {lu : LUniv}(c : Choice)
  (P : Process ∞ {lu} c)
  (nonDivP : NonDivergent P)
  → stableSch (nonDivBecomeStable1 c P nonDivP)
nonDivBecomeStable3Sch c (terminate x) nonDivP = _
nonDivBecomeStable3Sch c (node x) nonDivP = nonDivBecomeStable+3Sch c
  x nonDivP

```

--@END

mutual

```

nonDivBecomesStableBisimProof∞ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (nondiv' : NonDivergent∞ P)
  (a : ChoiceSet c)
  (terequivP : TerminateEquivalent∞ a P)
  → Bisimw (nonDivBecomeStable∞1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof∞ P nondiv' a terequivP =
  nonDivBecomesStableBisimProof (forceP P) (forceND nondiv') a terequivP

```

```

nonDivBecomesStableBisimProof : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (nondiv' : NonDivergent P)
  (a : ChoiceSet c)
  (terequivP : TerminateEquivalent a P)
  → Bisimw (nonDivBecomeStable1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof (terminate x) nondiv' .x termeqterm
  = BismwRef (terminate x)
nonDivBecomesStableBisimProof (node x) nondiv' a
  (termeqnode terequivP) =
  nonDivBecomesStableBisimProof+ x nondiv' a terequivP

```

```

nonDivBecomesStableBisimProof+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (nondiv' : NonDivergent+ P)
  (a : ChoiceSet c)
  (terequivP : TerminateEquivalent+ a P)
  → Bisimw (nonDivBecomeStable+1 c P nondiv') (terminate a)
nonDivBecomesStableBisimProof+ P (nondiv x (inj1 int)) a terequivP
  = nonDivBecomesStableBisimProof∞ (PI P int) (x int) a
    (onlyIntChoice terequivP int)
nonDivBecomesStableBisimProof+ P (nondiv x (inj2 stab)) a terequivP
  = eqterminater (termeqnode terequivP)

```

mutual

```

emptyTrPtoQImpliesEq      : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  (PS' : stable P)(tr : TrP {lu} [] (inj1 P') P)
  → P ≡ P'
emptyTrPtoQImpliesEq (terminate x) .(terminate x) pat (empty .x) = refl
emptyTrPtoQImpliesEq (node x) .(node x) PS' (tnode empty) = refl
emptyTrPtoQImpliesEq (node Q) P' PS'
  (tnode (intc .[]).(inj1 P') x1 x2)) = ⊥-elim (stabToNoInternal+ Q PS' x1) {- (PS'
emptyTrPtoQImpliesEq+      : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(P' : Process ∞ {lu} c)
  (PS' : stable+ P)(tr : TrP+ {lu} [] (inj1 P') P)
  → node P ≡ P'
emptyTrPtoQImpliesEq+ P P' PS' tr
  = emptyTrPtoQImpliesEq (node P) P' PS' (tnode tr)

```

--@BEGIN@bisimPPWithEmptyTr

mutual

```

-- this is the 4th component of nonDivBecomeStable
-- the statement we need is
-- all .. exists P'. traceproerty(P') /\ stableSch(P') /\ P bisimilar to P'
-- The labelled lemma should contain this as well:
-- but it should refer to stable and not stable sch
-- stablesch is just an intermediate step

-- the lemma is

```

```

-- all .. exists P'. traceproerty(P') /\ stable(P') /\ P bisimilar to P'

bisimPPWithEmptyTr∞ : {lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P') (PS' : stable∞ P')
  (nonDivP : NonDivergent∞ P)
  (tr : TrP∞ {lu} [])
  (inj1 (nonDivBecomeStable∞1 c P nonDivP)) P)
→ Bisimw (nonDivBecomeStable∞1 c P nonDivP)
  (forcep P')

bisimPPWithEmptyTr∞ P P' PP' PS' nonDivP tr =
  bisimPPWithEmptyTr (forcep P) (forcep P')
    (forceB PP') PS' (forceND nonDivP) tr

bisimPPWithEmptyTr : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P') (PS' : stable P')
  (nonDivP : NonDivergent P)
  (tr : TrP {lu} [] (inj1
    (nonDivBecomeStable1 {lu} c P nonDivP)) P)
→ Bisimw (nonDivBecomeStable1 {lu} c P nonDivP) P'

bisimPPWithEmptyTr {lu} {c} .(terminate x) (terminate x1)
  PP' PS' nonDivP (empty x) = PP'

bisimPPWithEmptyTr (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv nondivPI (inj1 x)) (tnode tr) =
  nonDivBecomesStableBisimProof∞ (PI P x)
  (nondivPI x) a (onlyIntChoice terequivP x)

bisimPPWithEmptyTr (node P) (terminate x)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv x1 (inj2 y)) (tnode tr) =
  eqterminater (termeqnode terequivP)

bisimPPWithEmptyTr (terminate P) (node P') PP' PS' nonDivP tr =
  PP'

bisimPPWithEmptyTr (node P) (node P') (eqnode bisimPP') PS'
  (nondiv x chemptyornot) (tnode tr) =
  bisimPPWithEmptyTr+ P P' bisimPP'
  PS' (nondiv x chemptyornot) tr

```

```

bisimPPWithEmptyTr+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P') (PS' : stable+ P')
  (nonDivP : NonDivergent+ P)
  (tr : TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable+1 c P nonDivP)
    (node P')
bisimPPWithEmptyTr+ {lu}{c} P P' PP' PS'
  (nondiv nondiv' (inj1 x1)) tr = PP''

where
  P'~ : Process∞ ∞ {lu} c
  P'~ = bisimIP' PP' x1

  trP'P'~ : P' →+* [ [] ] (forcep (P'~))
  trP'P'~ = bisimltr PP' x1

  P'≡P'~ : node P' ≡ forcep P'~ {∞}
  P'≡P'~ = emptyTrPtoQImpliesEq+ P'
    (forcep P'~) PS' trP'P'~

  P'~≡P' : forcep P'~ {∞} ≡ node P'
  P'~≡P' rewrite P'≡P'~ = refl

  P'~stable : stable (forcep P'~)
  P'~stable rewrite P'~≡P' = PS'

  PP'' : Bisimw (nonDivBecomeStable1 c
    (forcep (PI P x1)) (forceND (nondiv' x1)) )
    (forcep P'~)
  PP'' = bisimPPWithEmptyTr (forcep (PI P x1))
    (forcep P'~ {∞})
    (forceB (bisimInext PP' x1))
    P'~stable (forceND (nondiv' x1))
    (nonDivBecomeStable2 c
    (forcep (PI P x1)) (forceND (nondiv' x1)))

  PP''' : Bisimw (nonDivBecomeStable∞1 c
    (PI P x1) (nondiv' x1)) (node P')

```

PP'' **rewrite** P'≡P'~ = PP''

bisimPPWithEmptyTr+ P P' PP' PS'
 (nondiv x (inj₂ y)) empty = eqnode PP'
 bisimPPWithEmptyTr+ P P' PP' PS'
 (nondiv x (inj₂ y))
 (intc .[] .(inj₁
 (node P)) x₁ x₂) = eqnode PP'

--@END

mutual

choicesetornotBism : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)(PP' : B
 (cc' : ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P)))
 → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))

choicesetornotBism {c} P P' PP' (inj₁ ip) = inj₁ (bism2l PP' ip)
 choicesetornotBism {c} P P' PP' (inj₂ notip) = inj₂ (λ ip' → notip (bism2lr PP' ip'))

mutual

nondivLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
 → Bisims∞ {i} P P'
 → NonDivergent∞ {i} P → NonDivergent∞ {i} P'
 forceND (nondivLemBisims∞ P P' PP' nP) = nondivLemBisims (forcep P) (forcep P') (forcep P')

nondivLemBisims : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
 → Bisims {i} P P'
 → NonDivergent {i} P → NonDivergent {i} P'
 nondivLemBisims .(terminate a) .(terminate a) (eqterminate {a}) nP = tt
 nondivLemBisims .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') nP = nondivLemBisims+ Q Q'

nondivLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)

```

      → Bisims+ {i} P P'
      → NonDivergent+ {i} P → NonDivergent+ {i} P'
nondivLemBisims+ P P' PP' (nondiv f p) =
  nondiv (λ i → nondivLemBisims∞ (PI P (bisim2lr PP' i)) (PI P' i) (bisimlNext PP' i)
    (f (bisim2lr PP' i)) ) (choicesetornotBism P P' PP' p)

```

mutual

```

divLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → DivergentProcess∞ i c P → DivergentProcess∞ i c P'
divLemBisims∞ P P' PP' nP .forcediv = divLemBisims (forcep P) (forcep P') (forceB PP') (forcediv P)

divLemBisims : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → DivergentProcess i c P → DivergentProcess i c P'
divLemBisims .(terminate a) .(terminate a) (eqterminate {a}) nP = nP
divLemBisims .(node P) .(node P') (eqnode {P} {P'} PP') (div P divP) = div P' (divLemBisims+ P P' PP' nP)

```

```

divLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → DivergentProcess+ i c P → DivergentProcess+ i c P'
divLemBisims+ P P' PP' (div+ int Q) = div+ (bisim2l PP' int)
  (divLemBisims∞ (PI P int) (PI P' (bisim2l PP' int)) (bisimlNext PP' int) Q)

```

mutual

```

nondivLemBisims∞r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → NonDivergent∞ {i} P' → NonDivergent∞ {i} P
forceND (nondivLemBisims∞r P P' PP' nP) = nondivLemBisimsr (forcep P) (forcep P') (forceB PP') nP

nondivLemBisimsr : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → NonDivergent {i} P' → NonDivergent {i} P
nondivLemBisimsr .(terminate a) .(terminate a) (eqterminate {a}) nP = tt
nondivLemBisimsr .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') nP = nondivLemBisims+r Q Q' QQ' nP

```

```

nondivLemBisims+r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → NonDivergent+ {i} P' → NonDivergent+ {i} P
nondivLemBisims+r P P' PP' (nondiv f p) =
  nondiv (λ i → nondivLemBisims∞r (PI P i) ((PI P' (bisim2l PP' i))) (bisimlNext PP'
    (f (bisim2l PP' i))) (swapChoiceSetssr P P' PP' p)

```

mutual

```

divLemBisims∞r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → DivergentProcess∞ i c P' → DivergentProcess∞ i c P
divLemBisims∞r {i} P P' PP' nP .forcediv = divLemBisimsr (forcep P) (forcep P') (forceB P)
divLemBisimsr : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → DivergentProcess i c P' → DivergentProcess i c P
divLemBisimsr .(terminate a) .(terminate a) (eqterminate {a}) nP = nP
divLemBisimsr .(node P') .(node P) (eqnode {P'} {P} PP') (div P divP) = div P' (divLemBisimsr

```

```

divLemBisims+r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → DivergentProcess+ i c P' → DivergentProcess+ i c P
divLemBisims+r P P' PP' (div+ int Q) = div+ ((bisim2lr PP' int))
  ((divLemBisims∞r (PI P ((bisim2lr PP' int))) (PI P' int))

```

mutual

```

stabLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ P P'
  → stable∞ P' → stable∞ P
stabLemBisims∞ P P' PP' PS' = stabLemBisims (forcep P) (forcep P') (forceB PP') PS'
stabLemBisims : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims P P'
  → stable P' → stable P
stabLemBisims .(terminate a) .(terminate a) (eqterminate {a}) PS' = PS'

```

```
stabLemBisims .(node P) .(node P') (eqnode {P} {P'} PP') PS' = stabLemBisims+ P P' PP' PS'
```

```
stabLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → stable+ P' → stable+ P
```

```
stabLemBisims+ P P' PP' (PnoI ,, PNoTick) = (λ int → PnoI (bisim2I PP' int)) ,, (λ t → PNoTick
```

mutual

```
divergentImpliesNotTermEquiv+ : {lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent+ a P)
  (divP : DivergentProcess+ ∞ {lu} c P)
  → ⊥
```

```
divergentImpliesNotTermEquiv+ c P a terequivP (div+ int divP) =
  divergentImpliesNotTermEquiv∞ c (PI P int) a (onlyIntChoice terequivP int) divP
```

```
divergentImpliesNotTermEquiv∞ : {lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent∞ a P)
  (divP : DivergentProcess∞ ∞ {lu} c P)
  → ⊥
```

```
divergentImpliesNotTermEquiv∞ c P a terP divP = divergentImpliesNotTermEquiv c (forceP P) a terP
```

```
divergentImpliesNotTermEquiv : {lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent a P)
  (divP : DivergentProcess ∞ {lu} c P)
  → ⊥
```

```
divergentImpliesNotTermEquiv c .(node P) a (termeqnode terequivP) (div P divP) =
  divergentImpliesNotTermEquiv+ c P a terequivP
```

mutual

```

bisimImpliesDivergentPreserv $\infty$  : {lu : LUniv}{c : Choice} (P P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')
  (divP : DivergentProcess $\infty$   $\infty$  {lu} c P)
  → DivergentProcess $\infty$   $\infty$  {lu} c P'
forcediv (bisimImpliesDivergentPreserv $\infty$  c P P' PP' divP) =
  bisimImpliesDivergentPreserv c (forceP P) (forceP P') (forceB PP')
  (forcediv divP)

bisimImpliesDivergentPreserv+ : {lu : LUniv}{c : Choice} (P P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
  (divP : DivergentProcess+  $\infty$  {lu} c P)
  → DivergentProcess+  $\infty$  {lu} c P'
bisimImpliesDivergentPreserv+ c P P' PP' divP = bisimdiv PP' divP

--@BEGIN@bisimImpliesDivergentPreserv

bisimImpliesDivergentPreserv :
  {lu : LUniv}{c : Choice}
  (P P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  (divP : DivergentProcess  $\infty$  {lu} c P)
  → DivergentProcess  $\infty$  {lu} c P'
bisimImpliesDivergentPreserv c .(terminate  $\_$ ) P' (eqterminate x) ()
bisimImpliesDivergentPreserv c .(node P) .(terminate a) (eqterminater {a}
  {.(node P)} (termeqnode terequivP)) (div P divP)
  =  $\perp$ -elim (divergentImpliesNotTermEquiv+ c P a terequivP divP)
bisimImpliesDivergentPreserv c .(node P) .(node P') (eqnode {P} {P'} PP')
  (div P divP)
  = div P' (bisimImpliesDivergentPreserv+ c P P' PP' divP)

--@END

mutual
bisimStableImpliesNotDivergent $\infty$ ' : {lu : LUniv}{c : Choice} (P P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  P P')
  (PS' : stable $\infty$  P')
  → NonDivergent $\infty$  P
forceND (bisimStableImpliesNotDivergent $\infty$ ' c P P' PP' PS') =
  bisimStableImpliesNotDivergent' c (forceP P)

```

(forcep P') (forceB PP') PS'

bisimStableImpliesNotDivergent' : {lu : LUniv}{c : Choice} (P P' : Process ∞ {lu} c)
 (PP' : Bisimw P P')
 (PS' : stable P')
 → NonDivergent P

bisimStableImpliesNotDivergent' c (terminate x) P' PP' PS' = tt
 bisimStableImpliesNotDivergent' c (node P) .(terminate a)
 (eqterminater {a} terequiv) PS' =

TerImpliesNotDivergentaux c (node P) a terequiv

bisimStableImpliesNotDivergent' c (node P) .(node P')
 (eqnode {P} {P'} PP') PS' =
 bisimStableImpliesNotDivergent+' c P P' PP' PS'

bisimStableImpliesNotDivergent+' : {lu : LUniv}{c : Choice} (P P' : Process+ ∞ {lu} c)
 (PP' : Bisimw+ P P') (PS' : stable+ P')
 → NonDivergent+ P

bisimStableImpliesNotDivergent+' c P P' PP' PS' =
 nondiv+r PP' (nondiv (λ i → ⊥-elim (stabToNoInternal+ P' PS' i)) (inj₂ (s

lemBisimDRefusalAux : {lu : LUniv}{c : Choice}
 (x : ChoiceSet c)
 (P : Process+ ∞ {lu} c)
 (hasTauorTickNoTau : ChoiceSet (I P) ⊔ (¬ (ChoiceSet (I P)) × ChoiceSet (I P)) → ⊥)
 (PS : ChoiceSet (I P) → ⊥)
 → ChoiceSet (T P)

lemBisimDRefusalAux c x P (inj₁ x₁) PS = ⊥-elim (PS x₁)

lemBisimDRefusalAux c x P (inj₂ (−, x₁)) PS = x₁

mutual

bisimDRefusal+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c) (P' : Process+ ∞ {lu} c)
 (PS : stable+ P)
 (PP' : Bisimw+ {∞} P P')
 (X : Label lu → Bool)
 (isRoscoe : Bool)
 (ref : DRefusal+ P isRoscoe X)

```

→ DRefusal+ P' isRoscoe X
bisimDRefusal+ {c} P P' PS PP' X isRoscoe (drefusal noextChInX noTerm) =
  drefusal (bisimDRefusal+NoExtChInX P P' PS PP' X noextChInX)
  (bisimDRefusalNoTicksIfsRoscoe P P' PS PP' isRoscoe noTerm)

bisimDRefusalNoTicksIfsRoscoe : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PS : stable+ P)
  (PP' : Bisimw+ {∞} P P')
  (isRoscoe : Bool)
  (ref : NoTicksIfsRoscoe P isRoscoe)
  → NoTicksIfsRoscoe P' isRoscoe

bisimDRefusalNoTicksIfsRoscoe {lu} {c} P P' PS PP' isRoscoe ref tickIsIncl x =
  ref tickIsIncl (lem c P (PT P' x) p
    where
      path : TrP+ {lu} [] (inj₂ (PT P' x)) P
      path = bisimTtrr PP' x

```

```

lem : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c)(x : ChoiceSet c)
  (tr : TrP+ {lu} [] (inj₂ x) P)
  (PS : stable+ P) → ChoiceSet (T P)
lem c P x (intc .[] .(inj₂ x) x' x₂) PS = ⊥-elim (stabToNoInternal+ P PS x')
lem c P .(PT P x₁) (terc x₁) PS = x₁

```

```

bisimDRefusal+NoExtChInX : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PS : stable+ P)
  (PP' : Bisimw+ {∞} P P')
  (X : Label lu → Bool)
  (ref : NoExtChInX P X)
  → NoExtChInX P' X

bisimDRefusal+NoExtChInX {lu}{c} P P' PS PP' X ref e x = lem2 c P Q (Lab P' e) tr
  where
    Q : Process ∞ {lu} c
    Q = forcep (PP' .bisimEP'r e)

    tr : TrP+ {lu} (Lab P' e :: []) (inj₁ Q) P

```

$tr = PP' . \text{bisimEtrr} \quad e$

$\text{lem2} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process} \infty \{lu\} c)(Q : \text{Process} \infty \{lu\} c)$
 $(l : \text{Label } lu)(tr : \text{TrP} \{lu\} (l :: [])) (\text{inj}_1 Q) P)$
 $(PS : \text{stable} P)(X : \text{Label } lu \rightarrow \text{Bool})(ref : \text{NoExtChInX } P X)(labX : \text{True } (X l)) \rightarrow \perp$
 $\text{lem2 } c P Q . (\text{Lab } P x) (\text{extc } .[] . (\text{inj}_1 Q) x x_1) PS X ref labX = ref x labX$
 $\text{lem2 } c P Q l (\text{intc } .(l :: []) . (\text{inj}_1 Q) x x_1) PS X ref labX = \text{stabToNoInternal} P PS x$

$\text{bisimDRefusal} : \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c) (P' : \text{Process} \infty \{lu\} c)$
 $(PS : \text{stable } P)$
 $(PP' : \text{Bisimw } \{\infty\} P P')$
 $(X : \text{Label } lu \rightarrow \text{Bool})$
 $(isRoscoe : \text{Bool})$
 $(ref : \text{DRefusal } P isRoscoe X)$
 $\rightarrow \text{DRefusal } P' isRoscoe X$
 $\text{bisimDRefusal } (\text{terminate } x) (\text{terminate } x_1) PS PP' X isRoscoe ref tickIncl = ref tickIncl$
 $\text{bisimDRefusal } (\text{terminate } x) (\text{node } Q) PS (\text{eqterminate } (\text{termeqnode } terequivP)) X isRoscoe$
 $\text{drefusal } (\lambda e \rightarrow \perp\text{-elim } (\text{noExtChoice } terequivP e))(\lambda t \rightarrow \perp\text{-elim } ($
 $\text{bisimDRefusal } \{lu\}\{c\} (\text{node } P) (\text{terminate } x) PS$
 $(\text{eqterminater } (\text{termeqnode } terequivP)) X isRoscoe (\text{drefusal } noextChInX noTerm) tickInc$
 $= noTerm tickInc (\text{lemBisimDRefusalAux } c x P (\text{hasTauOrTickNoTau } terequivP) (\text{stabToNoInternal} P PS x))$
 $\text{bisimDRefusal } (\text{node } P) (\text{node } P') PS (\text{eqnode } bisimQQ') X isRoscoe ref =$
 $\text{bisimDRefusal} P P' PS bisimQQ' X isRoscoe ref$

mutual

$\text{lemmaxxx}_1 : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{divQ} : \text{DivergentProcess} \infty \{lu\} c Q)$
 $\rightarrow \text{BisimForNextP} \quad \text{result } (\text{inj}_1 Q)$
 $\rightarrow \text{Process} \infty \{lu\} c$
 $\text{lemmaxxx}_1 (\text{inj}_1 Q') Q \text{divQ BisimResultQ} = Q'$
 $\text{lemmaxxx}_1 (\text{inj}_2 y) Q \text{divQ BisimResultQ} = \perp\text{-elim } (\text{lemmaDivNotTermequiv } Q \text{divQ } y \text{ BisimResultQ})$
 $\text{lemmaxxx}_2 : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu))(P : \text{Process} \infty \{lu\} c)(Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{divQ} : \text{DivergentProcess} \infty \{lu\} c Q)$
 $\rightarrow (\text{bisimForNext} : \text{BisimForNextP} \quad \text{result } (\text{inj}_1 Q))$
 $\rightarrow (\text{trp} : \text{TrP } \{lu\} l \text{result } P)$

$\rightarrow \text{TrP } \{lu\} l (\text{inj}_1 (\text{lemmaxxx}_1 \text{ result } Q \text{ divQ bisimForNext})) P$
 $\text{lemmaxxx}_2 (\text{inj}_1 x) l P Q \text{ divQ bisimForNext trp} = \text{trp}$
 $\text{lemmaxxx}_2 (\text{inj}_2 y) l P Q \text{ divQ bisimForNext trp} = \perp\text{-elim } (\text{lemmaDivNotTermequiv } Q \text{ divQ } y \text{ bisimForNext})$

$\text{lemmaxxx}_2+ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu)) (P : \text{Process}+ \infty \{lu\} c) (Q : \text{Process } \infty \{lu\} c)$
 $\rightarrow (\text{divQ} : \text{DivergentProcess } \infty \{lu\} c Q)$
 $\rightarrow (\text{bisimForNext} : \text{BisimForNextP } \text{result } (\text{inj}_1 Q))$
 $\rightarrow (\text{trp}+ : \text{TrP}+ \{lu\} l \text{result } P)$
 $\rightarrow \text{TrP}+ \{lu\} l (\text{inj}_1 (\text{lemmaxxx}_1 \text{ result } Q \text{ divQ bisimForNext})) P$
 $\text{lemmaxxx}_2+ (\text{inj}_1 x) l P Q \text{ divQ bisimForNext trp}+ = \text{trp}+$
 $\text{lemmaxxx}_2+ (\text{inj}_2 y) l P Q \text{ divQ bisimForNext trp}+ = \perp\text{-elim } (\text{lemmaDivNotTermequiv } Q \text{ divQ } y \text{ bisimForNext})$

$\text{lemmaxxx}_3 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu)) (Q : \text{Process } \infty \{lu\} c)$
 $\rightarrow (\text{divQ} : \text{DivergentProcess } \infty \{lu\} c Q)$
 $\rightarrow (\text{bisimForNext} : \text{BisimForNextP } \text{result } (\text{inj}_1 Q))$
 $\rightarrow \text{Bisimw } (\text{lemmaxxx}_1 \text{ result } Q \text{ divQ bisimForNext}) Q$
 $\text{lemmaxxx}_3 (\text{inj}_1 Q') l Q \text{ divQ bisimForNext} = \text{bisimForNext}$
 $\text{lemmaxxx}_3 (\text{inj}_2 y) l Q \text{ divQ bisimForNext} = \perp\text{-elim } (\text{lemmaDivNotTermequiv } Q \text{ divQ } y \text{ bisimForNext})$

mutual

$\text{lemmayyy}_1 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process } \infty \{lu\} c)$
 $\rightarrow (\text{stab} : \text{stable } Q)$
 $\rightarrow (X : \text{Label } lu \rightarrow \text{Bool})$
 $\rightarrow \text{DRefusal } \{lu\} \{c\} Q \text{ true } X$
 $\rightarrow \text{BisimForNextP } \text{result } (\text{inj}_1 Q)$
 $\rightarrow \text{Process } \infty \{lu\} c$
 $\text{lemmayyy}_1 (\text{inj}_1 Q') Q \text{ stab } X x x_1 = Q'$
 $\text{lemmayyy}_1 (\text{inj}_2 y) Q \text{ stab } X \text{ dref termequiv } Q =$
 $\perp\text{-elim } (\text{lemmaDrefusalStableNotTermequiv } Q X \text{ dref stab } y \text{ termequiv } Q)$

$\text{lemmayyy}_1+ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process } \infty \{lu\} c)$
 $\rightarrow (\text{stab} : \text{stable } Q)$

```

→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true X
→ BisimForNextP result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1+ (inj1 Q') Q stab X x x1 = Q'
lemmayyy1+ (inj2 y) Q stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy2 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ TrP {lu} l result P
→ TrP {lu} l (inj1 (lemmayyy1 result Q stab X dref bisim)) P
lemmayyy2 (inj1 Q') l P Q stab X dref bisim tr = tr
lemmayyy2 (inj2 y) l P Q stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy2+ : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab+ : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ TrP+ {lu} l result P
→ TrP+ {lu} l (inj1 (lemmayyy1 result Q stab+ X dref bisim)) P
lemmayyy2+ (inj1 Q') l P Q stab X dref bisim tr = tr
lemmayyy2+ (inj2 y) l P Q stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy3 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))

```

```

→ Bisimw (lemmayyy1 result Q      stab X dref bisim) Q
lemmayyy3 (inj1 Q') Q      stab X dref bisim = bisim
lemmayyy3 (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemEqList : {A : Set} (l : List A) → l ++ [] ≡ l
lemEqList [] = refl
lemEqList (x :: l) = cong (λ l' → x :: l') (lemEqList l)

stableNotTerminateEquivaux : {lu : LUniv} {c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (hasTauOrTickNoTau : ChoiceSet (I P) ⊔
    (¬ (ChoiceSet (I P)) × ChoiceSet (T P)))
  (stabSch : stableSch+ P)
  (notick : noTickIfRoscoe+ true P)
  → ⊥

stableNotTerminateEquivaux P x (inj1 int) stabSch notick = stabSch int
stableNotTerminateEquivaux P x (inj2 (noint „ tick)) stabSch notick = notick tick

stableNotTerminateEquiv : {lu : LUniv} {c : Choice} (P : Process ∞ {lu} c)
  (x : ChoiceSet c)
  (terequiv : TerminateEquivalent x P)
  (stab : stable P)
  → ⊥

stableNotTerminateEquiv (terminate x) x1 terequiv stab = stab _
stableNotTerminateEquiv (node P) x (termeqnode terequivP) (stabSch „ notick)
  = stableNotTerminateEquivaux P x (hasTauOrTickNoTau terequivP) stabSch notick

noIntNoTerImpliesNoTermTrace : {lu : LUniv} {c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (tr : TrP+ [] (inj2 x) P)
  (noInt : ¬ (ChoiceSet (I P)))
  (noTer : ¬ (ChoiceSet (T P)))
  → ⊥

noIntNoTerImpliesNoTermTrace P x (intc .[] (inj2 x) int x2) noInt noTer = noInt int
noIntNoTerImpliesNoTermTrace P .(PT P ter') (terc ter') noInt noTer = noTer ter'

```

mutual

```

bisimwStableToNoTick : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (stabP' : stable P')
  (stabSchP : stableSch P)
  → noTickIfRoscoe true P

bisimwStableToNoTick (terminate x) P' (eqterminate terequiv) stabP' stabSchP =
  stableNotTerminateEquiv P' x terequiv
bisimwStableToNoTick (terminate x) (terminate _) (eqterminater terequiv) stabP' stabSchP = stabP'
bisimwStableToNoTick (node P) (terminate x) PP' stabP' stabSchP t = stabP' _
bisimwStableToNoTick (node P) (node P') (eqnode PP') (noInt „ noterP') noterP ter'
  = noIntNoTerImpliesNoTermTrace P' (PT P ter') (bisimTtr PP' ter') noInt noterP'

```

mutual

--@BEGIN@bisimRefusalros

```

bisimRefusalros : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P') (l : List (Label lu))
  (X : Label lu → Bool)
  (fail : failure P' l true X)
  → failure P l true X

bisimRefusalros {lu}{c} P P' PP' l X
  (stableFail (stableFp Q' tr' stab' drefuse'))
  = (stableFail (stableFp Qhat trhat2
    (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat stabNoTick )
    drefusehat))

where
  Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
  Qcom = bisimTraceTrP1 P P' PP' l (inj1 Q') tr'

  trcom : TrP {lu} l (bisimTraceTrP1 P P' PP' l
    (inj1 Q') tr') P
  trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') tr'

  QQ'com : BisimForNextP (bisimTraceTrP1 P P' PP' l
    (inj1 Q') tr') (inj1 Q')

```

$$QQ'com = bisimTraceTrP_3 \quad P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \quad tr'$$

$$\begin{aligned} Q &: \text{Process} \infty \{lu\} \ c \\ Q &= \text{lemmayyy}_1 \ Qcom \ Q' \quad stab' \ X \ drefuse' \ QQ'com \end{aligned}$$

$$\begin{aligned} tr &: \quad TrP \ \{lu\} \ l \ (\text{inj}_1 \ Q) \ P \\ tr &= \quad \text{lemmayyy}_2 \ Qcom \ l \ P \ Q' \quad stab' \ X \ drefuse' \ QQ'com \ trcom \end{aligned}$$

$$\begin{aligned} QQ' &: \text{Bisimw} \ Q \ Q' \\ QQ' &= \quad \text{lemmayyy}_3 \ Qcom \ Q' \quad stab' \ X \ drefuse' \ QQ'com \end{aligned}$$

$$\begin{aligned} Qhat &: \text{Process} \infty \{lu\} \ c \\ Qhat &= \text{nonDivBecomeStable}_1 \ c \ Q \\ &\quad (\text{bisimStableImpliesNotDivergent} \ c \ Q \ Q' \ QQ' \ stab' \\ &\quad (\text{stableImpliesNonDiv} \ Q' \ stab')) \end{aligned}$$

$$\begin{aligned} trhat &: \text{TrP} \ \{lu\} \ [] \ (\text{inj}_1 \ Qhat) \ Q \\ trhat &= \text{nonDivBecomeStable}_2 \ c \ Q \\ &\quad (\text{bisimStableImpliesNotDivergent} \ c \ Q \ Q' \ QQ' \ stab' \\ &\quad (\text{stableImpliesNonDiv} \ Q' \ stab')) \end{aligned}$$

$$\begin{aligned} QhatQ' &: \quad \text{Bisimw} \ Qhat \ Q' \\ QhatQ' &= \quad \text{bisimPPWithEmptyTr} \ Q \ Q' \ QQ' \ stab' \\ &\quad (\text{bisimStableImpliesNotDivergent} \ c \ Q \ Q' \ QQ' \ stab' \\ &\quad (\text{stableImpliesNonDiv} \ Q' \ stab')) \ trhat \end{aligned}$$

$$\begin{aligned} stabSchQhat &: \text{stableSch} \ Qhat \\ stabSchQhat &= \text{nonDivBecomeStable}_3 \text{Sch} \ c \ Q \\ &\quad (\text{bisimStableImpliesNotDivergent} \ c \ Q \ Q' \ QQ' \ stab' \\ &\quad (\text{stableImpliesNonDiv} \ Q' \ stab')) \end{aligned}$$

$$\begin{aligned} stabNoTick &: \text{noTickIfRoscoe} \ \text{true} \ Qhat \\ stabNoTick &= \text{bisimwStableToNoTick} \ Qhat \ Q' \ QhatQ' \ stab' \ stabSchQhat \end{aligned}$$

$$\begin{aligned} trhat_1 &: \text{TrP} \ \{lu\} \ (l \ ++ \ []) \ (\text{inj}_1 \ Qhat) \ P \\ trhat_1 &= \text{trPAppendTrw} \ c \ P \ Q \ l \quad [] \ (\text{inj}_1 \ Qhat) \ tr \ trhat \end{aligned}$$

$$\begin{aligned} eql &: \quad (l \ ++ \ []) \equiv l \\ eql &= \text{lemEqList} \ l \end{aligned}$$

$$trhat_2 : \text{TrP} \ \{lu\} \ l \ (\text{inj}_1 \ Qhat) \ P$$

○

○

$\text{trhat}_2 = \text{subst } (\lambda l' \rightarrow \text{TrP } \{lu\} l' (\text{inj}_1 \text{ Qhat}) P) \text{ eql trhat}_1$

$\text{drefusehat} : \text{DRefusal Qhat true } X$
 $\text{drefusehat} = \text{bisimDRefusal } Q' \text{ Qhat stab'}$
 $(\text{BismwSym Qhat } Q' \text{ QhatQ'}) X \text{ true drefuse'}$

$\text{bisimRefusalros } \{lu\}\{c\} P P' PP' l X$
 $(\text{divergentFailure } (\text{trdiv } Q' \text{ trp' divq'}))$
 $= (\text{divergentFailure } (\text{trdiv } Q \text{ tr divp}))$

where

$\text{Qcom} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1 P P' PP' l (\text{inj}_1 Q') \text{ trp'}$

$\text{trcom} : \text{TrP } \{lu\} l (\text{bisimTraceTrP}_1 P P' PP' l (\text{inj}_1 Q') \text{ trp'}) P$
 $\text{trcom} = \text{bisimTraceTrP}_2 P P' PP' l (\text{inj}_1 Q') \text{ trp'}$

$\text{QQ'com} : \text{BisimForNextP}$
 $(\text{bisimTraceTrP}_1 P P' PP' l (\text{inj}_1 Q') \text{ trp'}) (\text{inj}_1 Q')$
 $\text{QQ'com} = \text{bisimTraceTrP}_3 P P' PP' l (\text{inj}_1 Q') \text{ trp'}$

$Q : \text{Process } \infty \{lu\} c$
 $Q = \text{lemmaxxx}_1 \text{ Qcom } Q' \text{ divq' QQ'com}$

$\text{tr} : \text{TrP } \{lu\} l (\text{inj}_1 Q) P$
 $\text{tr} = \text{lemmaxxx}_2 \text{ Qcom } l P Q' \text{ divq' QQ'com trcom}$

$\text{QQ'} : \text{Bisimw } Q Q'$
 $\text{QQ'} = \text{lemmaxxx}_3 \text{ Qcom } l Q' \text{ divq' QQ'com}$

$Q'Q : \text{Bisimw } Q' Q$
 $Q'Q = \text{BismwSym } Q Q' QQ'$

$\text{divp} : \text{DivergentProcess } \infty \{lu\} c Q$
 $\text{divp} = \text{bisimImpliesDivergentPreserv } c Q' Q Q'Q \text{ divq'}$

--@END

--@BEGIN@bisimRefusalrosplus

mutual

○

○

```

bisimRefusalros+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (l : List (Label lu))
  (X : Label lu → Bool)
  (fail : failure+ P' l true      X)
  → failure+ P l true      X
bisimRefusalros+ {lu}{c} P P' PP' l X
  (stableFail (stableFp Q' tr' stab' drefuse'))
  = (stableFail (stableFp Qhat trhat2)
    (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat stabNoTick) drefuse')

```

where

```

Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
Qcom  = bisimTraceTrP1+ P P' PP' l (inj1 Q') tr'

trcom : TrP+ l (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') P
trcom = bisimTraceTrP2+ P P' PP' l (inj1 Q') tr'

QQ'com : BisimForNextP
        (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') (inj1 Q')
QQ'com = bisimTraceTrP3+ P P' PP' l (inj1 Q') tr'

Q : Process ∞ {lu} c
Q = lemmayyy1 Qcom Q'      stab' X drefuse' QQ'com

tr : TrP+ {lu} l (inj1 Q) P
tr = lemmayyy2+ Qcom l P Q' stab' X drefuse' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmayyy3 Qcom Q'      stab' X drefuse' QQ'com

Qhat : Process ∞ {lu} c
Qhat  = nonDivBecomeStable1 c Q
        (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
         (stableImpliesNonDiv Q' stab'))

trhat : TrP {lu} [] (inj1 Qhat) Q
trhat  = nonDivBecomeStable2 c Q
        (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
         (stableImpliesNonDiv Q' stab'))

```



$\text{QhatQ}' :$ $\text{Bisimw Qhat } Q'$
 $\text{QhatQ}' =$ $\text{bisimPPWithEmptyTr } Q \text{ } Q' \text{ } QQ' \text{ } stab'$
 $(\text{bisimStableImpliesNotDivergent } c \text{ } Q \text{ } Q' \text{ } QQ' \text{ } stab'$
 $(\text{stableImpliesNonDiv } Q' \text{ } stab')) \text{ } \text{trhat}$

$\text{stabSchQhat} : \text{stableSch Qhat}$
 $\text{stabSchQhat} = \text{nonDivBecomeStable}_3 \text{Sch } c \text{ } Q$
 $(\text{bisimStableImpliesNotDivergent } c \text{ } Q \text{ } Q' \text{ } QQ' \text{ } stab'$
 $(\text{stableImpliesNonDiv } Q' \text{ } stab'))$

$\text{stabNoTick} : \text{noTickIfRoscoe true Qhat}$
 $\text{stabNoTick} = \text{bisimwStableToNoTick Qhat } Q' \text{ } \text{QhatQ}' \text{ } stab' \text{ } \text{stabSchQhat}$

$\text{trhat}_1 : \text{TrP+ } \{lu\} (l ++ []) (\text{inj}_1 \text{ Qhat}) P$
 $\text{trhat}_1 = \text{trPAppendTrw+ } c \text{ } P \text{ } Q \text{ } l \quad [] (\text{inj}_1 \text{ Qhat}) \text{ } \text{tr trhat}$

$\text{eqI} : (l ++ []) \equiv l$
 $\text{eqI} = \text{lemEqList } l$

$\text{trhat}_2 : \text{TrP+ } \{lu\} l (\text{inj}_1 \text{ Qhat}) P$
 $\text{trhat}_2 = \text{subst } (\lambda l' \rightarrow \text{TrP+ } \{lu\} l' (\text{inj}_1 \text{ Qhat}) P) \text{ } \text{eqI trhat}_1$

$\text{drefusehat} :$ $\text{DRefusal Qhat true } X$
 $\text{drefusehat} =$ $\text{bisimDRefusal } Q' \text{ } \text{Qhat } stab'$
 $(\text{BismwSym Qhat } Q' \text{ } \text{QhatQ}') \text{ } X \text{ } \text{true drefuse'}$

$\text{bisimRefusalros+ } \{lu\} \{c\} P \text{ } P' \text{ } PP' \text{ } l \text{ } X$
 $(\text{divergentFailure } (\text{trdiv } Q' \text{ } \text{trp' } \text{divq'}))$
 $= (\text{divergentFailure } (\text{trdiv } Q \text{ } \text{tr divp}))$

where

$\text{Qcom} : \text{Process } \infty \{lu\} c \text{ } \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1 + \{lu\} P \text{ } P' \text{ } PP' \text{ } l (\text{inj}_1 \text{ } Q') \text{ } \text{trp'}$

$\text{trcom} : \text{TrP+ } \{lu\} l (\text{bisimTraceTrP}_1 + P \text{ } P' \text{ } PP' \text{ } l (\text{inj}_1 \text{ } Q') \text{ } \text{trp'}) P$
 $\text{trcom} = \text{bisimTraceTrP}_2 + P \text{ } P' \text{ } PP' \text{ } l (\text{inj}_1 \text{ } Q') \text{ } \text{trp'}$

$\text{QQ'com} : \text{BisimForNextP}$
 $(\text{bisimTraceTrP}_1 + P \text{ } P' \text{ } PP' \text{ } l (\text{inj}_1 \text{ } Q') \text{ } \text{trp'}) (\text{inj}_1 \text{ } Q')$



```

QQ'com = bisimTraceTrP3+          P P' PP' l (inj1 Q') trp'

Q : Process ∞ {lu} c
Q = lemmaxxx1 Qcom Q' divq' QQ'com

tr : TrP+ {lu} l (inj1 Q) P
tr = lemmaxxx2+ Qcom l P Q' divq' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmaxxx3 Qcom l Q' divq' QQ'com

Q'Q : Bisimw Q' Q
Q'Q = BismwSym Q Q' QQ'

divp : DivergentProcess ∞ {lu} c Q
divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divq'

```

```
--@END
```

```
--@BEGIN@bisimImFdiTwo
```

```

bisimImFDI2 : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
              (P' : Process ∞ {lu} c)
              (PP' : Bisimw {∞} P P')
              → P ⊑fdi2ros P'
bisimImFDI2 {lu}{c} P P' PP' = bisimRefusalros P P' PP'

bisimImFDI2r : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
              (P' : Process ∞ {lu} c)
              (PP' : Bisimw {∞} P P')
              → P' ⊑fdi2ros P
bisimImFDI2r {lu}{c} P P' PP' = bisimImFDI2 P' P (BismwSym P P' PP')

```

```
--@END
```

```

bisimImFDI2+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
              (P' : Process+ ∞ {lu} c)
              (PP' : Bisimw+ {∞} P P')
              → P ⊑fdi2ros+ P'

```

$$\text{bisimImFDI}_2 + \{lu\}\{c\} P P' PP' = \text{bisimRefusalros} + P P' PP'$$

$$\begin{aligned} \text{bisimImFDI}_2 r + : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} + \infty \{lu\} c) \\ (P' : \text{Process} + \infty \{lu\} c) \\ (PP' : \text{Bisimw} + \{\infty\} P P') \\ \rightarrow P' \sqsubseteq_{\text{fdi}_2 \text{ros}} + P \\ \text{bisimImFDI}_2 r + \{lu\}\{c\} P P' PP' = \text{bisimImFDI}_2 + P' P (\text{BisimwSym} + P P' PP') \end{aligned}$$

A.9 bisimilarityProofsWithSchneiderStable3.agda

```
--@PREFIX@bisimilarityProofs
```

```
module bisimilarityProofsWithSchneiderStable3 where
```

```
open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool      renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable
```

```
mutual
```

```

nondivImpliesIPornotIP : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (nondiv : NonDivergent+ P)
  → ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P))
nondivImpliesIPornotIP {c} P
  (nondiv - chemptyornot) = chemptyornot

```

mutual

```

swapChoiceSets : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (nondiv : NonDivergent+ P)
  → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))
swapChoiceSets {c} P P' PP' nond
  = nondivImpliesIPornotIP P' (nondiv+ PP' nond)

```

mutual

```

stableImpliesNonDiv∞ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (PS : stable∞ P) → NonDivergent∞ P
forceND (stableImpliesNonDiv∞ P PS) = stableImpliesNonDiv (forcep P) PS

stableImpliesNonDiv : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (PS : stable P)
  → NonDivergent P
stableImpliesNonDiv (terminate x) PS = tt
stableImpliesNonDiv (node x) PS = stableImpliesNonDiv+ x PS

stableImpliesNonDiv+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (PS : stable+ P) → NonDivergent+ P
stableImpliesNonDiv+ P PS = nondiv
  (stableImpliesNonDiv+aux P PS) (inj2 (stabToNoInternal+ P PS))

stableImpliesNonDiv+aux : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (PS : stable+ P)(i : ChoiceSet (I P))
  → NonDivergent∞ (PI P i)
stableImpliesNonDiv+aux P PS i = ⊥-elim (stabToNoInternal+ P PS i)

```

mutual

```

TerImpliesNotDivergentaux+ : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent+ a P)
  → NonDivergent+ P
TerImpliesNotDivergentaux+ c P a      terequiv = nondiv
  (λ i → TerImpliesNotDivergentaux∞ c
    (PI P i) a (onlyIntChoice terequiv i))
    (hasTauOrNotTau terequiv)

TerImpliesNotDivergentaux : {lu : LUniv}(c : Choice)(P : Process ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent a P)
  → NonDivergent P
TerImpliesNotDivergentaux c (terminate x) a terequiv = tt
TerImpliesNotDivergentaux c (node x) a (termeqnode terequivP)
  = TerImpliesNotDivergentaux+ c x a terequivP

TerImpliesNotDivergentaux∞ : {lu : LUniv}(c : Choice)(P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent a (forcep P))
  → NonDivergent∞ P
forceND (TerImpliesNotDivergentaux∞ c P a terequiv)
  = TerImpliesNotDivergentaux c (forcep P) a terequiv

```

```
--@BEGIN@bisimStableImpliesNotDivergent
```

mutual

```

bisimStableImpliesNotDivergent∞ : {lu : LUniv}(c : Choice)
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (PS' : stable∞ P')
  (nonDivP' : NonDivergent∞ P')
  → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞ c P P' PP' PS' nonDivP')
  = bisimStableImpliesNotDivergent c (forcep P)
    (forcep P')
    (forceB PP')

```

$$PS' (\text{forceND } \text{nonDivP}')$$

```

bisimStableImpliesNotDivergent : {lu : LUniv}(c : Choice)
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (PS' : stable P')
  (nonDivP' : NonDivergent P')
  → NonDivergent P
bisimStableImpliesNotDivergent c (terminate x) P' PP' PS' nonDivP' = tt
bisimStableImpliesNotDivergent c (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' nonDivP' =
  TerImpliesNotDivergentaux c (node P)
  a ((termeqnode terequivP))

```

```

bisimStableImpliesNotDivergent c (node P) (node P') (eqnode PP') PS'
  nonDivP' = bisimStableImpliesNotDivergent+
  c P P' PP' PS' nonDivP'

```

```

bisimStableImpliesNotDivergent+ : {lu : LUniv}(c : Choice)
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (PS' : stable+ P')
  (nonDivP' : NonDivergent+ P')
  → NonDivergent+ P
bisimStableImpliesNotDivergent+ c P P' PP' PS' nonDivP'
  = nondiv+r PP' nonDivP'

```

```
--@END
```

```
--@BEGIN@nonDivBecomeStable
```

```
mutual
```

```

nonDivBecomeStable∞1 : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (nonDivP : NonDivergent∞ P)
  → Process ∞ {lu} c
nonDivBecomeStable∞1 c P nonDivP = nonDivBecomeStable1
  c (forceP P) (forceND nonDivP)

```

```

nonDivBecomeStable $\infty_2$  : {lu : LUniv}(c : Choice)
  (P : Process $\infty$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $\infty$  P)
  → TrP $\infty$  {lu} [] (inj1
    (nonDivBecomeStable $\infty_1$  c P nonDivP)) P
nonDivBecomeStable $\infty_2$  c P nonDivP = nonDivBecomeStable2
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable $\infty_3$  : {lu : LUniv}(c : Choice)
  (P : Process $\infty$   $\infty$  {lu} c)
  (nonDivP : NonDivergent $\infty$  P)
  → stableSch (nonDivBecomeStable $\infty_1$ 
    c P nonDivP)
nonDivBecomeStable $\infty_3$  c P nonDivP = nonDivBecomeStable3
  c (forceP P) (forceND nonDivP)

nonDivBecomeStable+1 : {lu : LUniv}(c : Choice)
  (P : Process+  $\infty$  {lu} c)
  (nonDivP : NonDivergent+ P)
  → Process  $\infty$  {lu} c
nonDivBecomeStable+1 c P (nondiv x (inj1 int)) =
  nonDivBecomeStable $\infty_1$ 
    c (PI P int) (x int)
nonDivBecomeStable+1 c P (nondiv x (inj2 stab)) = node P

nonDivBecomeStable+2 : {lu : LUniv}(c : Choice)
  (P : Process+  $\infty$  {lu} c)
  (nonDivP : NonDivergent+ P)
  → TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P
nonDivBecomeStable+2 c P (nondiv x (inj1 int)) = intc [] (inj1
  (nonDivBecomeStable+1 c P
    (nondiv x (inj1 int)))) int
  (nonDivBecomeStable $\infty_2$  c
    (PI P int) (x int))
nonDivBecomeStable+2 c P (nondiv x (inj2 stab)) = empty

nonDivBecomeStable+3 : {lu : LUniv}(c : Choice)
  (P : Process+  $\infty$  {lu} c)
  (nonDivP : NonDivergent+ P)
  → stableSch -- stable

```

```

      (nonDivBecomeStable+1 c P nonDivP)
nonDivBecomeStable+3 c P (nondiv x (inj1 int)) =
      nonDivBecomeStable∞3 c
      (PI P int) (x int)
nonDivBecomeStable+3 c P (nondiv x (inj2 stab)) = stab -- , {!stab!} -- stab

nonDivBecomeStable1 : {lu : LUniv}{c : Choice}
      (P : Process ∞ {lu} c)
      (nonDivP : NonDivergent P)
      → Process ∞ {lu} c
nonDivBecomeStable1 c (terminate x) nonDivP = terminate x
nonDivBecomeStable1 c (node x) nonDivP =
      nonDivBecomeStable+1 c x nonDivP

nonDivBecomeStable2 : {lu : LUniv}{c : Choice}
      (P : Process ∞ {lu} c)
      (nonDivP : NonDivergent P)
      → TrP {lu} [] (inj1
      (nonDivBecomeStable1 c P nonDivP)) P
nonDivBecomeStable2 c (terminate x) nonDivP = empty x
nonDivBecomeStable2 c (node x) nonDivP
      = tnode (nonDivBecomeStable+2 c x nonDivP)

nonDivBecomeStable3 : {lu : LUniv}{c : Choice}
      (P : Process ∞ {lu} c)
      (nonDivP : NonDivergent P)
      → stableSch (nonDivBecomeStable1 c P nonDivP)
nonDivBecomeStable3 c (terminate x) nonDivP = _
nonDivBecomeStable3 c (node x) nonDivP = nonDivBecomeStable+3 c x nonDivP --
--
--@END

mutual
  nonDivBecomesStableBisimProof∞ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
      (nondiv' : NonDivergent∞ P)
      (a : ChoiceSet c)
      (terequivP : TerminateEquivalent∞ a P)

```

$\rightarrow \text{Bisimw } (\text{nonDivBecomeStable}_{\infty 1} \ c \ P \ \text{nondiv}') \ (\text{terminate } a)$
 $\text{nonDivBecomesStableBisimProof}_{\infty} \ P \ \text{nondiv}' \ a \ \text{terequiv} P =$
 $\text{nonDivBecomesStableBisimProof } (\text{forcep } P) \ (\text{forceND } \text{nondiv}') \ a \ \text{terequiv} P$

$\text{nonDivBecomesStableBisimProof} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process } \infty \ \{lu\} \ c)$
 $(\text{nondiv}' : \text{NonDivergent } P)$
 $(a : \text{ChoiceSet } c)$
 $(\text{terequiv} P : \text{TerminateEquivalent } a \ P)$
 $\rightarrow \text{Bisimw } (\text{nonDivBecomeStable}_1 \ c \ P \ \text{nondiv}') \ (\text{terminate } a)$
 $\text{nonDivBecomesStableBisimProof } (\text{terminate } x) \ \text{nondiv}' \ .x \ \text{termeqterm}$
 $= \text{BismwRef } (\text{terminate } x)$
 $\text{nonDivBecomesStableBisimProof } (\text{node } x) \ \text{nondiv}' \ a$
 $(\text{termeqnode } \text{terequiv} P) =$
 $\text{nonDivBecomesStableBisimProof+ } x \ \text{nondiv}' \ a \ \text{terequiv} P$

$\text{nonDivBecomesStableBisimProof+} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+ } \infty \ \{lu\} \ c)$
 $(\text{nondiv}' : \text{NonDivergent+ } P)$
 $(a : \text{ChoiceSet } c)$
 $(\text{terequiv} P : \text{TerminateEquivalent+ } a \ P)$
 $\rightarrow \text{Bisimw } (\text{nonDivBecomeStable+}_1 \ c \ P \ \text{nondiv}') \ (\text{terminate } a)$
 $\text{nonDivBecomesStableBisimProof+ } P \ (\text{nondiv } x \ (\text{inj}_1 \ \text{int})) \ a \ \text{terequiv} P$
 $= \text{nonDivBecomesStableBisimProof}_{\infty} \ (\text{PI } P \ \text{int}) \ (x \ \text{int}) \ a$
 $(\text{onlyIntChoice } \text{terequiv} P \ \text{int})$
 $\text{nonDivBecomesStableBisimProof+ } P \ (\text{nondiv } x \ (\text{inj}_2 \ \text{stab})) \ a \ \text{terequiv} P$
 $= \text{eqterminater } (\text{termeqnode } \text{terequiv} P)$

mutual

$\text{emptyTrPtoQImpliesEq} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process } \infty \ \{lu\} \ c)$
 $(PS' : \text{stable } P) (tr : \text{TrP } \{lu\} \ \square \ (\text{inj}_1 \ P') \ P)$
 $\rightarrow P \equiv P'$

$\text{emptyTrPtoQImpliesEq } (\text{terminate } x) \ .(\text{terminate } x) \ \text{pat } (\text{empty } .x) = \text{refl}$
 $\text{emptyTrPtoQImpliesEq } (\text{node } x) \ .(\text{node } x) \ PS' \ (\text{tnode } \text{empty}) = \text{refl}$
 $\text{emptyTrPtoQImpliesEq } (\text{node } Q) \ P' \ PS'$

$(\text{tnode } (\text{intc } \square \ .(\text{inj}_1 \ P') \ x_1 \ x_2)) = \perp\text{-elim } (\text{stabToNoInternal+ } Q \ PS' \ x_1) \ \{- (PS' \ x_1) \}$

$\text{emptyTrPtoQImpliesEq+} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+ } \infty \ \{lu\} \ c) (P' : \text{Process } \infty \ \{lu\} \ c)$
 $(PS' : \text{stable+ } P) (tr : \text{TrP+ } \{lu\} \ \square \ (\text{inj}_1 \ P') \ P)$
 $\rightarrow \text{node } P \equiv P'$

emptyTrPtoQImpliesEq+ $P P' PS' tr$
 $= \text{emptyTrPtoQImpliesEq } (\text{node } P) P' PS' (\text{tnode } tr)$

--@BEGIN@bisimPPWithEmptyTr

mutual

bisimPPWithEmptyTr ∞ : $\{lu : LUniv\}\{c : Choice\}$
 $(P P' : Process\infty \infty \{lu\} c)$
 $(PP' : Bisimw\infty P P') (PS' : stable\infty P')$
 $(nonDivP : NonDivergent\infty P)$
 $(tr : TrP\infty \{lu\} [])$
 $(inj_1 (nonDivBecomeStable\infty_1 c P nonDivP)) P)$
 $\rightarrow Bisimw (nonDivBecomeStable\infty_1 c P nonDivP)$
 $(forceP P')$

bisimPPWithEmptyTr $\infty P P' PP' PS' nonDivP tr =$
 $\text{bisimPPWithEmptyTr } (forceP P) (forceP P')$
 $(forceB PP') PS' (forceND nonDivP) tr$

bisimPPWithEmptyTr : $\{lu : LUniv\}\{c : Choice\}$
 $(P P' : Process \infty \{lu\} c)$
 $(PP' : Bisimw P P') (PS' : stable P')$
 $(nonDivP : NonDivergent P)$
 $(tr : TrP \{lu\} []) (inj_1$
 $(nonDivBecomeStable_1 \{lu\} c P nonDivP)) P)$
 $\rightarrow Bisimw (nonDivBecomeStable_1 \{lu\} c P nonDivP) P'$

bisimPPWithEmptyTr $\{lu\} \{c\} .(terminate x) (terminate x_1)$
 $PP' PS' nonDivP (\text{empty } x) = PP'$

bisimPPWithEmptyTr $(\text{node } P) (terminate a)$
 $(eqterminater (termeqnode terequivP))$
 $PS' (nondiv nondivPI (inj_1 x)) (\text{tnode } tr) =$
 $\text{nonDivBecomesStableBisimProof}\infty (PI P x)$
 $(nondivPI x) a (\text{onlyIntChoice } terequivP x)$

bisimPPWithEmptyTr $(\text{node } P) (terminate x)$
 $(eqterminater (termeqnode terequivP))$
 $PS' (nondiv x_1 (inj_2 y)) (\text{tnode } tr) =$
 $\text{eqterminater } (termeqnode terequivP)$

bisimPPWithEmptyTr $(terminate P) (\text{node } P') PP' PS' nonDivP tr =$
 PP'

```

bisimPPWithEmptyTr (node P) (node P') (eqnode bisimPP') PS'
  (nondiv x chemptyornot) (tnode tr) =
  bisimPPWithEmptyTr+ P P' bisimPP'
  PS' (nondiv x chemptyornot) tr

```

```

bisimPPWithEmptyTr+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P') (PS' : stable+ P')
  (nonDivP : NonDivergent+ P)
  (tr : TrP+ {lu} [] (inj1
    (nonDivBecomeStable+1 c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable+1 c P nonDivP)
    (node P')
bisimPPWithEmptyTr+ {lu}{c} P P' PP' PS'
  (nondiv nondiv' (inj1 x1)) tr = PP''

```

where

```

P'~ : Process∞ ∞ {lu} c
P'~ = bisimIP' PP' x1

```

```

trP'P'~ : P' →+*[ [] ] (forcep (P'~))
trP'P'~ = bisimltr PP' x1

```

```

P'≡P'~ : node P' ≡ forcep P'~ {∞}
P'≡P'~ = emptyTrPtoQImpliesEq+ P'
  (forcep P'~) PS' trP'P'~

```

```

P'~≡P' : forcep P'~ {∞} ≡ node P'
P'~≡P' rewrite P'≡P'~ = refl

```

```

P'~stable : stable (forcep P'~)
P'~stable rewrite P'~≡P' = PS'

```

```

PP'' : Bisimw (nonDivBecomeStable1 c
  (forcep (PI P x1)) (forceND (nondiv' x1)) )
  (forcep P'~)
PP'' = bisimPPWithEmptyTr (forcep (PI P x1))
  (forcep P'~ {∞})
  (forceB (bisimlnext PP' x1))

```

```

      P'~stable (forceND (nondiv' x1))
    (nonDivBecomeStable2 c
      (forcep (PI P x1)) (forceND (nondiv' x1)))

    PP''' : Bisimw (nonDivBecomeStable∞1 c
      (PI P x1) (nondiv' x1)) (node P')
    PP''' rewrite P'≡P'~ = PP''

  bisimPPWithEmptyTr+ P P' PP' PS'
    (nondiv x (inj2 y)) empty = eqnode PP'
  bisimPPWithEmptyTr+ P P' PP' PS'
    (nondiv x (inj2 y))
    (intc .[] .(inj1
      (node P)) x1 x2) = eqnode PP'

--@END

mutual
  choicestornotBism : {i : Size}{lu : LUniv}{c : Choice}{P P' : Process+ ∞ {lu} c}(PP' : B
    (cc' : ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P)))
    → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))

  choicestornotBism {c} P P' PP' (inj1 ip) = inj1 (bisim2l PP' ip)
  choicestornotBism {c} P P' PP' (inj2 notip) = inj2 (λ ip' → notip (bisim2lr PP' ip'))

mutual
  nondivLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}{P P' : Process∞ ∞ {lu} c}
    → Bisims∞ {i} P P'
    → NonDivergent∞ {i} P → NonDivergent∞ {i} P'
  forceND (nondivLemBisims∞ P P' PP' nP) = nondivLemBisims (forcep P) (forcep P') (forcep P')

  nondivLemBisims : {i : Size}{lu : LUniv}{c : Choice}{P P' : Process ∞ {lu} c}
    → Bisims {i} P P'
    → NonDivergent {i} P → NonDivergent {i} P'
  nondivLemBisims .(terminate a) .(terminate a) (eqterminate {a}) nP = tt
  nondivLemBisims .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') nP = nondivLemBisims+ Q

```

```

nondivLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → NonDivergent+ {i} P → NonDivergent+ {i} P'
nondivLemBisims+ P P' PP' (nondiv f p) =
  nondiv (λ i → nondivLemBisims∞ (PI P (bisim2lr PP' i)) (PI P' i) (bisimlNext PP' i)
    (f (bisim2lr PP' i)) (choicesetornotBism P P' PP' p))

```

mutual

```

divLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → DivergentProcess∞ i c P → DivergentProcess∞ i c P'
divLemBisims∞ P P' PP' nP .forcediv = divLemBisims (forcep P) (forcep P') (forceB PP') (forcediv P)

divLemBisims : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → DivergentProcess i c P → DivergentProcess i c P'
divLemBisims .(terminate a) .(terminate a) (eqterminate {a}) nP = nP
divLemBisims .(node P) .(node P') (eqnode {P} {P'} PP') (div P divP) = div P' (divLemBisims+ P P' PP')

```

```

divLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → DivergentProcess+ i c P → DivergentProcess+ i c P'
divLemBisims+ P P' PP' (div+ int Q) = div+ (bisim2l PP' int)
  (divLemBisims∞ (PI P int) (PI P' (bisim2l PP' int)) (bisimlNext PP' int) Q)

```

mutual

```

nondivLemBisims∞r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → NonDivergent∞ {i} P' → NonDivergent∞ {i} P
forceND (nondivLemBisims∞r P P' PP' nP) = nondivLemBisimsr (forcep P) (forcep P') (forceB PP') (forcediv P)

nondivLemBisimsr : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → NonDivergent {i} P' → NonDivergent {i} P

```

```

nondivLemBisimsr .(terminate a) .(terminate a) (eqterminate {a}) nP = tt
nondivLemBisimsr .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') nP = nondivLemBisims+r

```

```

nondivLemBisims+r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → NonDivergent+ {i} P' → NonDivergent+ {i} P
nondivLemBisims+r P P' PP' (nondiv f p) =
  nondiv (λ i → nondivLemBisims∞r (PI P i) ((PI P' (bisim2l PP' i))) (bisimlNext PP' i)
    (f (bisim2l PP' i))) (swapChoiceSetssr P P' PP' p)

```

mutual

```

divLemBisims∞r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → DivergentProcess∞ i c P' → DivergentProcess∞ i c P
divLemBisims∞r {i} P P' PP' nP .forcediv = divLemBisimsr (forcep P) (forcep P') (forceB PP' nP)

divLemBisimsr : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → DivergentProcess i c P' → DivergentProcess i c P

```

```

divLemBisimsr .(terminate a) .(terminate a) (eqterminate {a}) nP = nP
divLemBisimsr .(node P') .(node P) (eqnode {P'} {P} PP') (div P divP) = div P' (divLemBisimsr P' PP')

```

```

divLemBisims+r : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → DivergentProcess+ i c P' → DivergentProcess+ i c P
divLemBisims+r P P' PP' (div+ int Q) = div+ ((bisim2lr PP' int))
  ((divLemBisims∞r (PI P ((bisim2lr PP' int))) (PI P' int)) ((bisim2lr PP' int)))

```

mutual

```

stabLemBisims∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ P P'
  → stable∞ P' → stable∞ P
stabLemBisims∞ P P' PP' PS' = stabLemBisims (forcep P) (forcep P') (forceB PP') PS'

stabLemBisims : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)

```

```

      → Bisims P P'
      → stable P' → stable P
stabLemBisims .(terminate a) .(terminate a) (eqterminate {a}) PS' = PS'
stabLemBisims .(node P) .(node P') (eqnode {P} {P'} PP') PS' = stabLemBisims+ P P' PP' PS'

stabLemBisims+ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → stable+ P' → stable+ P
stabLemBisims+ P P' PP' (PnoI ,, PNoTick) = (λ int → PnoI (bisim2l PP' int)) ,, (λ t → PNoTick)

```

mutual

```

divergentImpliesNotTermEquiv+ : {lu : LUniv}(c : Choice)
  (P : Process+ ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent+ a P)
  (divP : DivergentProcess+ ∞ {lu} c P)
  → ⊥

divergentImpliesNotTermEquiv+ c P a terequivP (div+ int divP) =
  divergentImpliesNotTermEquiv∞ c (PI P int) a (onlyIntChoice terequivP int) divP

divergentImpliesNotTermEquiv∞ : {lu : LUniv}(c : Choice)
  (P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent∞ a P)
  (divP : DivergentProcess∞ ∞ {lu} c P)
  → ⊥

divergentImpliesNotTermEquiv∞ c P a terP divP = divergentImpliesNotTermEquiv c (forcep P) a terP

divergentImpliesNotTermEquiv : {lu : LUniv}(c : Choice)
  (P : Process ∞ {lu} c)
  (a : ChoiceSet c)
  (terP : TerminateEquivalent a P)
  (divP : DivergentProcess ∞ {lu} c P)
  → ⊥

divergentImpliesNotTermEquiv c .(node P) a (termeqnode terequivP) (div P divP) =
  divergentImpliesNotTermEquiv+ c P a terequivP divP

```

mutual

```

bisimImpliesDivergentPreserv $\infty$  : {lu : LUniv}(c : Choice) (P P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')
  (divP : DivergentProcess $\infty$   $\infty$  {lu} c P)
  → DivergentProcess $\infty$   $\infty$  {lu} c P'
forcediv (bisimImpliesDivergentPreserv $\infty$  c P P' PP' divP) =
  bisimImpliesDivergentPreserv c (forceP P) (forceP P') (forceB PP')
  (forcediv divP)

```

```

bisimImpliesDivergentPreserv+ : {lu : LUniv}(c : Choice) (P P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
  (divP : DivergentProcess+  $\infty$  {lu} c P)
  → DivergentProcess+  $\infty$  {lu} c P'
bisimImpliesDivergentPreserv+ c P P' PP' divP = bisimdiv PP' divP

```

--@BEGIN@bisimImpliesDivergentPreserv

```

bisimImpliesDivergentPreserv : {lu : LUniv}(c : Choice)
  (P P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  (divP : DivergentProcess  $\infty$  {lu} c P)
  → DivergentProcess  $\infty$  {lu} c P'
bisimImpliesDivergentPreserv c .(terminate _) P' (eqterminate x) ()
bisimImpliesDivergentPreserv c .(node P) .(terminate a) (eqterminater {a}
  {.(node P)}) (termeqnode terequivP)) (div P divP)
  =  $\perp$ -elim (divergentImpliesNotTermEquiv+ c P a terequivP divP)
bisimImpliesDivergentPreserv c .(node P) .(node P') (eqnode {P} {P'} PP')
  (div P divP)
  = div P' (bisimImpliesDivergentPreserv+ c P P' PP' divP)

```

--@END

mutual

```

bisimStableImpliesNotDivergent $\infty$ ' : {lu : LUniv}(c : Choice) (P P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  P P')
  (PS' : stable $\infty$  P')

```

$$\begin{aligned} & \rightarrow \text{NonDivergent} \infty P \\ \text{forceND} \ (\text{bisimStableImpliesNotDivergent} \infty' \ c \ P \ P' \ PP' \ PS') = \\ & \quad \text{bisimStableImpliesNotDivergent}' \ c \ (\text{forceP} \ P) \\ & \quad (\text{forceP} \ P') \ (\text{forceB} \ PP') \ PS' \end{aligned}$$

$$\begin{aligned} \text{bisimStableImpliesNotDivergent}' : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P \ P' : \text{Process} \infty \{lu\} \ c) \\ (PP' : \text{Bisimw} \ P \ P') \\ (PS' : \text{stable} \ P') \\ \rightarrow \text{NonDivergent} \ P \end{aligned}$$

$$\begin{aligned} \text{bisimStableImpliesNotDivergent}' \ c \ (\text{terminate} \ x) \ P' \ PP' \ PS' &= \text{tt} \\ \text{bisimStableImpliesNotDivergent}' \ c \ (\text{node} \ P) \ .(\text{terminate} \ a) \\ & \quad (\text{eqterminater} \ \{a\} \ \text{terequiv}) \ PS' = \\ & \quad \text{TerImpliesNotDivergentaux} \ c \ (\text{node} \ P) \ a \ \text{terequiv} \\ \text{bisimStableImpliesNotDivergent}' \ c \ (\text{node} \ P) \ .(\text{node} \ P') \\ & \quad (\text{eqnode} \ \{.P\} \ \{P'\} \ PP') \ PS' = \\ & \quad \text{bisimStableImpliesNotDivergent}+' \ c \ P \ P' \ PP' \ PS' \end{aligned}$$

$$\begin{aligned} \text{bisimStableImpliesNotDivergent}+' : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P \ P' : \text{Process}+ \infty \{lu\} \ c) \\ (PP' : \text{Bisimw}+ \ P \ P') \ (PS' : \text{stable}+ \ P') \\ \rightarrow \text{NonDivergent}+ \ P \end{aligned}$$

$$\begin{aligned} \text{bisimStableImpliesNotDivergent}+' \ c \ P \ P' \ PP' \ PS' = \\ \text{nondiv}+r \ PP' \ (\text{nondiv} \ (\lambda \ i \rightarrow \perp\text{-elim} \ (\text{stabToNoInternal}+ \ P' \ PS' \ i)) \ (\text{inj}_2 \ (\text{stab} \end{aligned}$$

$$\begin{aligned} \text{lemBisimDRefusalAux} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (x : \text{ChoiceSet} \ c) \\ (P : \text{Process}+ \infty \{lu\} \ c) \\ (\text{hasTauorTickNoTau} : \text{ChoiceSet} \ (\text{I} \ P) \ \uplus \ (\neg \ (\text{ChoiceSet} \ (\text{I} \ P)) \times \text{ChoiceSet} \\ (PS : \text{ChoiceSet} \ (\text{I} \ P) \rightarrow \perp) \\ \rightarrow \text{ChoiceSet} \ (\text{T} \ P) \end{aligned}$$

$$\text{lemBisimDRefusalAux} \ c \ x \ P \ (\text{inj}_1 \ x_1) \ PS = \perp\text{-elim} \ (PS \ x_1)$$

$$\text{lemBisimDRefusalAux} \ c \ x \ P \ (\text{inj}_2 \ (-, x_1)) \ PS = x_1$$

mutual

$$\begin{aligned} \text{bisimDRefusal}+ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P : \text{Process}+ \infty \{lu\} \ c) \ (P' : \text{Process}+ \infty \{lu\} \ c) \\ (PS : \text{stable}+ \ P) \\ (PP' : \text{Bisimw}+ \ \{\infty\} \ P \ P') \end{aligned}$$

```

(X : Label lu → Bool)
(isRoscoe : Bool)
(ref : DRefusal+ P isRoscoe X)
→ DRefusal+ P' isRoscoe X
bisimDRefusal+ {c} P P' PS PP' X isRoscoe (drefusal noextChInX noTerm) =
  drefusal (bisimDRefusal+NoExtChInX P P' PS PP' X noextChInX)
  (bisimDRefusalNoTicksIfsRoscoe P P' PS PP' isRoscoe noTerm)

```

```

bisimDRefusalNoTicksIfsRoscoe : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PS : stable+ P)
  (PP' : Bisimw+ {∞} P P')
  (isRoscoe : Bool)
  (ref : NoTicksIfsRoscoe P isRoscoe)
  → NoTicksIfsRoscoe P' isRoscoe
bisimDRefusalNoTicksIfsRoscoe {lu} {c} P P' PS PP' isRoscoe ref tickIsIncl x =
  ref tickIsIncl (lem c P (PT P' x) path)
  where
    path : TrP+ {lu} [] (inj2 (PT P' x)) P
    path = bisimTtrr PP' x

```

```

lem : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c)(x : ChoiceSet c)
  (tr : TrP+ {lu} [] (inj2 x) P)
  (PS : stable+ P) → ChoiceSet (T P)
lem c P x (intc .[] .(inj2 x) x' x2) PS = ⊥-elim (stabToNoInternal+ P PS x')
lem c P .(PT P x1) (terc x1) PS = x1

```

```

bisimDRefusal+NoExtChInX : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PS : stable+ P)
  (PP' : Bisimw+ {∞} P P')
  (X : Label lu → Bool)
  (ref : NoExtChInX P X)
  → NoExtChInX P' X
bisimDRefusal+NoExtChInX {lu}{c} P P' PS PP' X ref e x = lem2 c P Q (Lab P' e) tr PS
  where
    Q : Process ∞ {lu} c

```



$Q = \text{forcep } (PP' . \text{bisimEP}'r \ e)$

$\text{tr} : \text{TrP+ } \{lu\} (\text{Lab } P' \ e :: []) (\text{inj}_1 \ Q) \ P$
 $\text{tr} = PP' . \text{bisimEtrr} \quad e$

$\text{lem2} : \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process+ } \infty \{lu\} \ c)(Q : \text{Process } \infty \{lu\} \ c)$
 $(l : \text{Label } lu)(tr : \text{TrP+ } \{lu\} (l :: []) (\text{inj}_1 \ Q) \ P)$
 $(PS : \text{stable+ } P)(X : \text{Label } lu \rightarrow \text{Bool})(ref : \text{NoExtChInX } P \ X)(labX : \text{True } (X \ l)) \rightarrow \perp$
 $\text{lem2 } c \ P \ Q . (\text{Lab } P \ x) (\text{extc } [] . (\text{inj}_1 \ Q) \ x \ x_1) \ PS \ X \ ref \ labX = ref \ x \ labX$
 $\text{lem2 } c \ P \ Q \ l (\text{intc } .(l :: []) . (\text{inj}_1 \ Q) \ x \ x_1) \ PS \ X \ ref \ labX = \text{stabToNoInternal+ } P \ PS \ x$

$\text{bisimDRefusal} : \{lu : \text{LUniv}\}\{c : \text{Choice}\} (P : \text{Process } \infty \{lu\} \ c) (P' : \text{Process } \infty \{lu\} \ c)$
 $(PS : \text{stable } P)$
 $(PP' : \text{Bisimw } \{\infty\} \ P \ P')$
 $(X : \text{Label } lu \rightarrow \text{Bool})$
 $(isRoscoe : \text{Bool})$
 $(ref : \text{DRefusal } P \ isRoscoe \ X)$
 $\rightarrow \text{DRefusal } P' \ isRoscoe \ X$

$\text{bisimDRefusal } (\text{terminate } x) (\text{terminate } x_1) \ PS \ PP' \ X \ isRoscoe \ ref \ tickIncl = ref \ tickIncl$

$\text{bisimDRefusal } (\text{terminate } x) (\text{node } Q) \ PS \quad (\text{eqterminate } (\text{termeqnode } terequivP)) \ X \ isRoscoe \ ref =$
 $\text{drefusal } (\lambda \ e \rightarrow \perp\text{-elim } (\text{noExtChoice } terequivP \ e))(\lambda \ t \rightarrow \perp\text{-elim } (ref \ t))$

$\text{bisimDRefusal } \{lu\}\{c\} (\text{node } P) (\text{terminate } x) \ PS$
 $(\text{eqterminater } (\text{termeqnode } terequivP)) \ X \ isRoscoe (\text{drefusal } \text{noextChInX } \text{noTerm}) \ tickInc$
 $= \text{noTerm } tickInc (\text{lemBisimDRefusalAux } c \ x \ P (\text{hasTauOrTickNoTau } terequivP) (\text{stabToNoInternal+ } P \ P' \ PS \ bisimQQ' \ X \ isRoscoe \ ref))$
 $\text{bisimDRefusal } (\text{node } P) (\text{node } P') \ PS \quad (\text{eqnode } bisimQQ') \ X \ isRoscoe \ ref =$
 $\text{bisimDRefusal+ } P \ P' \ PS \ bisimQQ' \ X \ isRoscoe \ ref$

mutual

$\text{lemmaxxx}_1 : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} \ c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process } \infty \{lu\} \ c)$
 $\rightarrow (\text{divQ} : \text{DivergentProcess } \infty \{lu\} \ c \ Q)$
 $\rightarrow \text{BisimForNextP} \quad \text{result } (\text{inj}_1 \ Q)$
 $\rightarrow \text{Process } \infty \{lu\} \ c$

$\text{lemmaxxx}_1 (\text{inj}_1 \ Q') \ Q \ \text{divQ} \ \text{BisimResultQ} = Q'$

$\text{lemmaxxx}_1 (\text{inj}_2 \ y) \ Q \ \text{divQ} \ \text{BisimResultQ} = \perp\text{-elim } (\text{lemmaDivNotTermequiv } Q \ \text{divQ} \ y \ \text{BisimResultQ})$

$\text{lemmaxxx}_2 : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process } \infty \{lu\} \ c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu))(P : \text{Process } \infty \{lu\} \ c)(Q : \text{Process } \infty \{lu\} \ c)$



```

→ (divQ : DivergentProcess ∞ {lu} c Q)
→ (bisimForNext : BisimForNextP result (inj1 Q))
→ (trp : TrP {lu} l result P)
→ TrP {lu} l (inj1 (lemmaxxx1 result Q divQ bisimForNext)) P
lemmaxxx2 (inj1 x) l P Q divQ bisimForNext trp = trp
lemmaxxx2 (inj2 y) l P Q divQ bisimForNext trp = ⊥-elim (lemmaDivNotTermequiv Q div

```

```

lemmaxxx2+ : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (divQ : DivergentProcess ∞ {lu} c Q)
→ (bisimForNext : BisimForNextP result (inj1 Q))
→ (trp+ : TrP+ {lu} l result P)
→ TrP+ {lu} l (inj1 (lemmaxxx1 result Q divQ bisimForNext)) P
lemmaxxx2+ (inj1 x) l P Q divQ bisimForNext trp+ = trp+
lemmaxxx2+ (inj2 y) l P Q divQ bisimForNext trp+ = ⊥-elim (lemmaDivNotTermequiv Q

```

```

lemmaxxx3 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (l : List (Label lu))(Q : Process ∞ {lu} c)
→ (divQ : DivergentProcess ∞ {lu} c Q)
→ (bisimForNext : BisimForNextP result (inj1 Q))
→ Bisimw (lemmaxxx1 result Q divQ bisimForNext) Q
lemmaxxx3 (inj1 Q') l Q divQ bisimForNext = bisimForNext
lemmaxxx3 (inj2 y) l Q divQ bisimForNext = ⊥-elim (lemmaDivNotTermequiv Q divQ y bi

```

mutual

```

lemmayyy1 : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true X
→ BisimForNextP result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1 (inj1 Q') Q stab X x x1 = Q'
lemmayyy1 (inj2 y) Q stab X dref termequivQ =
⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy1+ : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (X : Label lu → Bool)
  → DRefusal {lu}{c} Q true X
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c
lemmayyy1+ (inj1 Q') Q      stab X x x1 = Q'
lemmayyy1+ (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy2 : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (X : Label lu → Bool)
  → (dref : DRefusal {lu}{c} Q true X)
  → (bisim : BisimForNextP result (inj1 Q))
  → TrP {lu} l result P
  → TrP {lu} l (inj1 (lemmayyy1 result Q      stab X dref bisim)) P
lemmayyy2 (inj1 Q') l P Q      stab X dref bisim tr = tr
lemmayyy2 (inj2 y) l P Q      stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy2+ : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab+ : stable Q)
  → (X : Label lu → Bool)
  → (dref : DRefusal {lu}{c} Q true X)
  → (bisim : BisimForNextP result (inj1 Q))
  → TrP+ {lu} l result P
  → TrP+ {lu} l (inj1 (lemmayyy1 result Q      stab+ X dref bisim)) P
lemmayyy2+ (inj1 Q') l P Q      stab X dref bisim tr = tr
lemmayyy2+ (inj2 y) l P Q      stab X dref termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

lemmayyy3 : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)

```

```

→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ Bisimw (lemmayyy1 result Q stab X dref bisim) Q
lemmayyy3 (inj1 Q') Q      stab X dref bisim = bisim
lemmayyy3 (inj2 y) Q      stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemEqList : {A : Set}{l : List A} → l ++ [] ≡ l
lemEqList [] = refl
lemEqList (x :: l) = cong (λ l' → x :: l') (lemEqList l)

```

```

stableNotTerminateEquivaux : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (hasTauOrTickNoTau : ChoiceSet (I P) ⊕
    (¬ (ChoiceSet (I P)) × ChoiceSet (T P)))
  (stab : stable+ P)
  (notick : noTickIfRoscoe+ true P)
  → ⊥

```

```

stableNotTerminateEquivaux P x (inj1 int) (stabsch „ x2) notick = stabsch int
stableNotTerminateEquivaux P x (inj2 (noint „ tick)) stab notick = notick tick

```

```

stableNotTerminateEquiv : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (x : ChoiceSet c)
  (terequiv : TerminateEquivalent x P)
  (stab : stable P)
  → ⊥

```

```

stableNotTerminateEquiv (terminate x) x1 terequiv stab = stab _
stableNotTerminateEquiv (node P) x (termeqnode terequivP) (stabSch „ notick)
  = stableNotTerminateEquivaux P x (hasTauOrTickNoTau terequivP) (stabSch „ notick) no

```

```

noIntNoTerImpliesNoTermTrace : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (tr : TrP+ [] (inj2 x) P)
  (noInt : ¬ (ChoiceSet (I P)))
  (noTer : ¬ (ChoiceSet (T P)))

```

```

→ ⊥
noIntNoTerImpliesNoTermTrace P x (intc .[] .(inj2 x) int x2) noInt noTer = noInt int
noIntNoTerImpliesNoTermTrace P .(PT P ter') (terc ter') noInt noTer = noTer ter'

```

mutual

```

bisimwStableToNoTick : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (stabP' : stable P')
  (stabP : stable P)
  → noTickIfRoscoe true P
bisimwStableToNoTick (terminate x) P' (eqterminate terequiv) stabP' stabP =
  stableNotTerminateEquiv P' x terequiv s
bisimwStableToNoTick (terminate x) .(terminate _) (eqterminater terequiv) stabP' stabP = stabP' _
bisimwStableToNoTick (node P) (terminate x) PP' stabP' stabP t = stabP' _
bisimwStableToNoTick (node P) (node P') (eqnode PP') (noInt „ noterP') noterP ter'
  = noIntNoTerImpliesNoTermTrace P' (PT P ter') (bisimTtr PP' ter') noInt noterP'

```

A.10 bisimilarityProofsWithSchneiderStable3Part2.agda

```
--@PREFIX@bisimilarityProofsWithSchneiderStablethreeParttwo
```

```
module bisimilarityProofsWithSchneiderStable3Part2 where
```

```

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction

```

```

open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable3
open import bisimSym
open import bisimilarityProofsWithSchneiderStable

mutual
  schStabNoTraceToInj2+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
    (stabSchP : stableSch+ P)
    (a : ChoiceSet c)
    (tr : TrP+ [] (inj2 a) P)
    (notick : ¬ (ChoiceSet (T P)))
    → ⊥
  schStabNoTraceToInj2+ P stabSchP a (intc .[] .(inj2 a) tauP tr) = ⊥-elim (stabSchP tauP)
  schStabNoTraceToInj2+ P stabSchP .(PT P x) (terc x) y = y x

mutual

--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartOne

stabSchBisim2stabRosclsStabRosc : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (stabP' : stable P')
  (stabSchP : stableSch P)
  → stable P

stabSchBisim2stabRosclsStabRosc
  (terminate x) (terminate x1)
  (eqterminate terequiv) stabP' stabSchP = stabP'
stabSchBisim2stabRosclsStabRosc
  (terminate x) (node P')
  (eqterminate (termeqnode terequivP'))

```

```

(P' stabSch „ notickP') stabSchP _
= tauOrTickNoTauP'ImpliesConclusion tauOrTickNoTauP' where
  tauOrTickNoTauP' : ChoiceSet (I P') ⊔ (¬ (ChoiceSet (I P')))
    × ChoiceSet (T P')
  tauOrTickNoTauP' = hasTauOrTickNoTau terequivP'

  tauOrTickNoTauP'ImpliesConclusion : ChoiceSet (I P')
    ⊔ (¬ (ChoiceSet (I P')))
    × ChoiceSet (T P') → ⊥

  tauOrTickNoTauP'ImpliesConclusion (inj₁ tauChoiceP')
    = P' stabSch tauChoiceP'
  tauOrTickNoTauP'ImpliesConclusion (inj₂ (¬ „ tickChoiceP'))
    = notickP' tickChoiceP'

```

```

stabSchBisim2stabRosclsStabRosc

```

```

  (terminate x) .(terminate _) (eqterminater terequiv)
  stabP' stabSchP
= stabP'

```

```

stabSchBisim2stabRosclsStabRosc

```

```

  (node P) (terminate x) PP' stabP' stabSchP
= ⊥-elim (stabP' _)

```

```

stabSchBisim2stabRosclsStabRosc

```

```

  (node P) (node P') (eqnode bisimQQ') (stabSchP' „ noTickP')
  stabSchP
= stabSchP „ noTickP

```

```

where

```

```

traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj₂ (PT P t)) P'
traceToTickP = bisimTtr bisimQQ'

```

```

noTickP : ¬ (ChoiceSet (T P))

```

```

noTickP t = schStabNoTraceToInj2+ P' stabSchP' (PT P t)
  (traceToTickP t) noTickP'

```

```

--@END

```

```

stabSchBisim2stabRosclsStabRosc+ : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (stabP' : stable+ P')
  (stabSchP : stableSch+ P)

```

$\rightarrow \text{stable}^+ P$

$\text{stabSchBisim2stabRosclsStabRosc}^+ P P' PP' (\text{stabSchP}' \text{ ,, } \text{noTickP}') \text{ stabSchP} = \text{stabSchP}$

where

$\text{traceToTickP} : (t : \text{ChoiceSet } (\text{T } P)) \rightarrow \text{TrP}^+ [] (\text{inj}_2 (\text{PT } P t)) P'$

$\text{traceToTickP} = \text{bisimTtr } PP'$

$\text{noTickP} : \neg (\text{ChoiceSet } (\text{T } P))$

$\text{noTickP } t = \text{schStabNoTraceToInj2}^+ P' \text{ stabSchP}' (\text{PT } P t) (\text{traceToTickP } t) \text{ noTickP}'$

$\text{stabSchBisim2stabRosclsStabRosc}^\infty : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P P' : \text{Process}^\infty \infty \{lu\} c)$
 $(PP' : \text{Bisimw}^\infty P P')$
 $(\text{stabP}' : \text{stable}^\infty P')$
 $(\text{stabSchP} : \text{stableSch}^\infty P)$
 $\rightarrow \text{stable}^\infty P$

$\text{stabSchBisim2stabRosclsStabRosc}^\infty P P' PP' \text{ stabP}' \text{ stabSchP} = \text{stabSchBisim2stabRosclsS}$
 $(\text{forcep } P) (\text{forcep } P') (t)$

mutual

$\text{lemmaDrefusalStableNotTermequiv}' : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (Q : \text{Process} \infty \{lu\} c)$
 $(\text{stab} : \text{stable } Q)$
 $(y : \text{ChoiceSet } c)$
 $(\text{termequivQ} : \text{TerminateEquivalent } y Q)$
 $\rightarrow \perp$

$\text{lemmaDrefusalStableNotTermequiv}' (\text{terminate } x) \text{ stab } y \text{ termequiv} = \perp\text{-elim } (\text{stab } \text{tt})$

$\text{lemmaDrefusalStableNotTermequiv}' (\text{node } Q)$
 $\text{stab } y (\text{termegnode } \text{terequivP}) = \text{hasTauOrTickGivesBot } \text{hasTauO}$

where

$\text{hasTauOrTickNoTau}' : \text{ChoiceSet } (\text{I } Q) \uplus$
 $\neg (\text{ChoiceSet } (\text{I } Q)) \times \text{ChoiceSet } (\text{T } Q)$

$\text{hasTauOrTickNoTau}' = \text{hasTauOrTickNoTau } \text{terequivP}$

$\text{hasTauOrTickGivesBot} : \text{ChoiceSet } (\text{I } Q) \uplus \neg (\text{ChoiceSet } (\text{I } Q)) \times \text{ChoiceSet } (\text{T } Q) \rightarrow \perp$

$\text{hasTauOrTickGivesBot } (\text{inj}_1 x) = \text{stabToNoInternal}^+ Q \text{ stab } x$

$\text{hasTauOrTickGivesBot } (\text{inj}_2 (\text{noti } \text{ ,, } t)) = (\text{proj}_2' \text{ stab}) t$

mutual

```

lemmayyy1' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c
lemmayyy1' (inj1 Q') Q      stab x1 = Q'
lemmayyy1' (inj2 y) Q      stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q      stab y termequivQ)

```

{- lemmayyy1∞' : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)

```

lemmayyy1+' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c
lemmayyy1+' (inj1 Q') Q      stab x1 = Q'
lemmayyy1+' (inj2 y) Q      stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q      stab y termequivQ)

```

```

lemmayyy2' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
  → (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (bisim : BisimForNextP result (inj1 Q))
  → TrP {lu} l result P
  → TrP {lu} l (inj1 (lemmayyy1' result Q      stab bisim)) P
lemmayyy2' (inj1 Q') l P Q      stab bisim tr = tr
lemmayyy2' (inj2 y) l P Q      stab termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q      stab y termequivQ)

```

```

lemmayyy2+' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
  → (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab+ : stable Q)
  → (bisim : BisimForNextP result (inj1 Q))

```

```

→ TrP+ {lu} l result P
→ TrP+ {lu} l (inj1 (lemmayyy1' result Q stab+ bisim)) P
lemmayyy2+' (inj1 Q') l P Q      stab bisim tr = tr
lemmayyy2+' (inj2 y) l P Q stab termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

lemmayyy3' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (bisim : BisimForNextP result (inj1 Q))
→ Bisimw (lemmayyy1' result Q stab bisim) Q
lemmayyy3' (inj1 Q') Q      stab bisim = bisim
lemmayyy3' (inj2 y) Q      stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartTwo

module _ {lu : LUniv}{c : Choice}(P P' : Process∞ ∞ c)
  (PP' : Bisimw∞ {∞} P P')(l : List (Label lu))
  (Q' : Process ∞ {lu} c)
  (tr' : TrP∞ l (inj1 Q') P')
  (stab' : stable Q') where

bisimTraceTrP∞Qcom : Process ∞ c ⊔ ChoiceSet c
bisimTraceTrP∞Qcom = bisimTraceTrP∞1 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞trcom : TrP∞ {lu} l (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') P
bisimTraceTrP∞trcom = bisimTraceTrP∞2 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞QQ'com : BisimForNextP (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') (inj1 Q')
bisimTraceTrP∞QQ'com = bisimTraceTrP∞3 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞Q : Process ∞ {lu} c

```

```

bisimTraceTrP∞Q = lemmayyy1' bisimTraceTrP∞Qcom Q' stab'
                        bisimTraceTrP∞QQ'com

```

```

bisimTraceTrP∞tr : TrP∞ {lu} l (inj1 bisimTraceTrP∞Q) P
bisimTraceTrP∞tr = lemmayyy2' (bisimTraceTrP∞Qcom) l (forcep P)
                        Q' stab' bisimTraceTrP∞QQ'com
                        bisimTraceTrP∞trcom

```

```

bisimTraceTrP∞QQ' : Bisimw bisimTraceTrP∞Q Q'
bisimTraceTrP∞QQ' = lemmayyy3'
                        bisimTraceTrP∞Qcom Q' stab'
                        bisimTraceTrP∞QQ'com

```

```

bisimTraceTrP∞Qhat : Process ∞ {lu} c
bisimTraceTrP∞Qhat = nonDivBecomeStable1 c bisimTraceTrP∞Q
                        (bisimStableImpliesNotDivergent c
                         bisimTraceTrP∞Q Q'
                         bisimTraceTrP∞QQ' stab'
                         (stableImpliesNonDiv Q' stab'))

```

```

bisimTraceTrP∞trhat : TrP {lu} [] (inj1 bisimTraceTrP∞Qhat)
                        bisimTraceTrP∞Q
bisimTraceTrP∞trhat = nonDivBecomeStable2 c
                        bisimTraceTrP∞Q
                        ( bisimStableImpliesNotDivergent c
                         bisimTraceTrP∞Q Q'
                         bisimTraceTrP∞QQ' stab'
                         (stableImpliesNonDiv Q' stab'))

```

```

bisimTraceTrP∞QhatQ' : Bisimw bisimTraceTrP∞Qhat Q'
bisimTraceTrP∞QhatQ' = bisimPPWithEmptyTr bisimTraceTrP∞Q Q'

```



```

bisimTraceTrP $\infty$ QQ' stab'
(  bisimStableImpliesNotDivergent c
  bisimTraceTrP $\infty$ Q Q'
  bisimTraceTrP $\infty$ QQ' stab'
  (stableImpliesNonDiv Q' stab'))
bisimTraceTrP $\infty$ trhat

```

```

bisimTraceTrP $\infty$ stabSchQhat : stableSch bisimTraceTrP $\infty$ Qhat
bisimTraceTrP $\infty$ stabSchQhat = nonDivBecomeStable3 c bisimTraceTrP $\infty$ Q
(  bisimStableImpliesNotDivergent c
  bisimTraceTrP $\infty$ Q Q'
  bisimTraceTrP $\infty$ QQ' stab'
  (stableImpliesNonDiv Q' stab'))

```

```

bisimTraceTrP $\infty$ stabQhat : stable bisimTraceTrP $\infty$ Qhat
bisimTraceTrP $\infty$ stabQhat = stabSchBisim2stabRosclsStabRosc
  bisimTraceTrP $\infty$ Qhat
  Q' bisimTraceTrP $\infty$ QhatQ' stab'
  bisimTraceTrP $\infty$ stabSchQhat

```

```

bisimTraceTrP $\infty$ trhat1 : TrP $\infty$  {lu} (l ++ [])
                        (inj1 bisimTraceTrP $\infty$ Qhat) P
bisimTraceTrP $\infty$ trhat1 = trPAppendTrw $\infty$  c P bisimTraceTrP $\infty$ Q l []
                        (inj1 bisimTraceTrP $\infty$ Qhat)
                        bisimTraceTrP $\infty$ tr bisimTraceTrP $\infty$ trhat

```

```

bisimTraceTrP $\infty$ trhat2 : TrP $\infty$  {lu} l (inj1 bisimTraceTrP $\infty$ Qhat) P
bisimTraceTrP $\infty$ trhat2 = subst (λ l' → TrP $\infty$  {lu} l'
                                (inj1 bisimTraceTrP $\infty$ Qhat)
                                P)
                                eql bisimTraceTrP $\infty$ trhat1 where

```

```

eql :      (l ++ [] ) ≡ l
eql = lemEqList l

```

```
--@END
```



A.11 bisimilarityProofsWithSchneiderStable3Part2Theorem

```
--@PREFIX@bisimilarityProofsWithSchneiderStablethreeParttwoTheo
```

```
module bisimilarityProofsWithSchneiderStable3Part2TheoremOnly where
```

```
open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool      renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable3
open import bisimSym
open import bisimilarityProofsWithSchneiderStable
```

```
mutual
```

```
schStabNoTraceToInj2+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (stabSchP : stableSch+ P)
  (a : ChoiceSet c)
  (tr : TrP+ [] (inj2 a) P)
  (notick : ¬ (ChoiceSet (T P)))
  → ⊥
```

```
schStabNoTraceToInj2+ P stabSchP a (intc .[] .(inj2 a) tauP tr) = ⊥-elim (stabSchP tauP)
```

```
schStabNoTraceToInj2+ P stabSchP .(PT P x) (terc x) y = y x
```

```
mutual
```

```
--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartOne
```

```
stabSchBisim2stabRosclsStabRosc : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (stabP' : stable P')
  (stabSchP : stableSch P)
  → stable P
```

```
stabSchBisim2stabRosclsStabRosc (terminate x) (terminate x1) (eqterminate terequiv) stabP'
stabSchBisim2stabRosclsStabRosc (terminate x) (node P') (eqterminate (termeqnode terequiv
  (P' stabSch „ notickP') stabSchP _ = tauOrTickNoTauP'ImpliesConclusion tauOr
tauOrTickNoTauP' : ChoiceSet (I P') ⊔ (¬ (ChoiceSet (I P')) × ChoiceSet (T P'))
tauOrTickNoTauP' = hasTauOrTickNoTau terequivP'
```

```
tauOrTickNoTauP'ImpliesConclusion : ChoiceSet (I P') ⊔ (¬ (ChoiceSet (I P')) × ChoiceSet (T P'))
tauOrTickNoTauP'ImpliesConclusion (inj1 tauChoiceP') = P' stabSch tauChoiceP'
tauOrTickNoTauP'ImpliesConclusion (inj2 ( _ „ tickChoiceP')) = notickP' tickChoiceP'
```

```
stabSchBisim2stabRosclsStabRosc (terminate x) .(terminate _) (eqterminater terequiv) stabP
stabSchBisim2stabRosclsStabRosc (node P) (terminate x) PP' stabP' stabSchP = ⊥-elim (st
stabSchBisim2stabRosclsStabRosc (node P) (node P') (eqnode bisimQQ') (stabSchP' „ noT
```

```
where
```

```
traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj2 (PT P t)) P'
traceToTickP = bisimTtr bisimQQ'
```

```
noTickP : ¬ (ChoiceSet (T P))
noTickP t = schStabNoTraceToInj2+ P' stabSchP' (PT P t) (traceToTickP t) noTickP'
```

```
--@END
```

```
stabSchBisim2stabRosclsStabRosc+ : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (stabP' : stable+ P')
  (stabSchP : stableSch+ P)
  → stable+ P
```

```

stabSchBisim2stabRosclsStabRosc+ P P' PP' (stabSchp' „ noTickP') stabSchP = stabSchP „ noTickP
  where
traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj2 (PT P t)) P'
traceToTickP = bisimTtr PP'

noTickP : ¬ (ChoiceSet (T P))
noTickP t = schStabNoTraceToInj2+ P' stabSchp' (PT P t) (traceToTickP t) noTickP'

```

```

stabSchBisim2stabRosclsStabRosc∞ : {lu : LUniv}{c : Choice}{P P' : Process∞ ∞ {lu} c}
  (PP' : Bisimw∞ P P')
  (stabP' : stable∞ P')
  (stabSchP : stableSch∞ P)
  → stable∞ P
stabSchBisim2stabRosclsStabRosc∞ P P' PP' stabP' stabSchP = stabSchBisim2stabRosclsStabRosc
  (forcep P) (forcep P') (forceB

```

mutual

```

lemmaDrefusalStableNotTermequiv' : {lu : LUniv}{c : Choice}{Q : Process ∞ {lu} c}
  (stab : stable Q)
  (y : ChoiceSet c)
  (termequivQ : TerminateEquivalent y Q)
  → ⊥

lemmaDrefusalStableNotTermequiv' (terminate x) stab y termequiv = ⊥-elim (stab tt)
lemmaDrefusalStableNotTermequiv' (node Q)
  stab y (termegnode terequivP) = hasTauOrTickGivesBot hasTauOrTickNo
  where
hasTauOrTickNoTau' : ChoiceSet (I Q) ⊔
  ¬ (ChoiceSet (I Q)) × ChoiceSet (T Q)
hasTauOrTickNoTau' = hasTauOrTickNoTau terequivP

hasTauOrTickGivesBot : ChoiceSet (I Q) ⊔ ¬ (ChoiceSet (I Q)) × ChoiceSet (T Q) → ⊥
hasTauOrTickGivesBot (inj1 x) = stabToNoInternal+ Q stab x
hasTauOrTickGivesBot (inj2 (noti „ t)) = (proj2' stab) t

```



mutual

$\text{lemmayyy}_1' : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{stab} : \text{stable } Q)$
 $\rightarrow \text{BisimForNextP } \text{result } (\text{inj}_1 Q)$
 $\rightarrow \text{Process} \infty \{lu\} c$
 $\text{lemmayyy}_1' (\text{inj}_1 Q') Q \text{ stab } x_1 = Q'$
 $\text{lemmayyy}_1' (\text{inj}_2 y) Q \text{ stab } \text{termequiv} Q =$
 $\perp\text{-elim } (\text{lemmaDrefusalStableNotTermequiv}' Q \text{ stab } y \text{ termequiv} Q)$

$\text{lemmayyy}_1+' : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{stab} : \text{stable } Q)$
 $\rightarrow \text{BisimForNextP } \text{result } (\text{inj}_1 Q)$
 $\rightarrow \text{Process} \infty \{lu\} c$
 $\text{lemmayyy}_1+' (\text{inj}_1 Q') Q \text{ stab } x_1 = Q'$
 $\text{lemmayyy}_1+' (\text{inj}_2 y) Q \text{ stab } \text{termequiv} Q =$
 $\perp\text{-elim } (\text{lemmaDrefusalStableNotTermequiv}' Q \text{ stab } y \text{ termequiv} Q)$

$\text{lemmayyy}_2' : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu)) (P : \text{Process} \infty \{lu\} c) (Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{stab} : \text{stable } Q)$
 $\rightarrow (\text{bisim} : \text{BisimForNextP } \text{result } (\text{inj}_1 Q))$
 $\rightarrow \text{TrP } \{lu\} l \text{ result } P$
 $\rightarrow \text{TrP } \{lu\} l (\text{inj}_1 (\text{lemmayyy}_1' \text{ result } Q \text{ stab } \text{bisim})) P$
 $\text{lemmayyy}_2' (\text{inj}_1 Q') l P Q \text{ stab } \text{bisim } tr = tr$
 $\text{lemmayyy}_2' (\text{inj}_2 y) l P Q \text{ stab } \text{termequiv} Q x =$
 $\perp\text{-elim } (\text{lemmaDrefusalStableNotTermequiv}' Q \text{ stab } y \text{ termequiv} Q)$

$\text{lemmayyy}_2+' : \{lu : \text{LUniv}\}\{c : \text{Choice}\} \rightarrow (\text{result} : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\rightarrow (l : \text{List } (\text{Label } lu)) (P : \text{Process}+ \infty \{lu\} c) (Q : \text{Process} \infty \{lu\} c)$
 $\rightarrow (\text{stab}+ : \text{stable } Q)$
 $\rightarrow (\text{bisim} : \text{BisimForNextP } \text{result } (\text{inj}_1 Q))$
 $\rightarrow \text{TrP}+ \{lu\} l \text{ result } P$
 $\rightarrow \text{TrP}+ \{lu\} l (\text{inj}_1 (\text{lemmayyy}_1' \text{ result } Q \text{ stab}+ \text{bisim})) P$



```

lemmayyy2+' (inj1 Q') l P Q stab bisim tr = tr
lemmayyy2+' (inj2 y) l P Q stab termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

lemmayyy3' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊔ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (bisim : BisimForNextP result (inj1 Q))
  → Bisimw (lemmayyy1' result Q stab bisim) Q
lemmayyy3' (inj1 Q') Q stab bisim = bisim
lemmayyy3' (inj2 y) Q stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartTwo

module _ {lu : LUniv}{c : Choice}
  (P P' : Process∞ ∞ c)
  (PP' : Bisimw∞ {∞} P P')(l : List (Label lu))
  (Q' : Process ∞ {lu} c)
  (tr' : TrP∞ l (inj1 Q') P')
  (stab' : stable Q') where
--@HIDE-BEG
bisimTraceTrP∞Qcom : Process ∞ c ⊔ ChoiceSet c
bisimTraceTrP∞Qcom = bisimTraceTrP∞1 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞trcom : TrP∞ {lu} l (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') P
bisimTraceTrP∞trcom = bisimTraceTrP∞2 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞QQ'com : BisimForNextP (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') (inj1 Q')
bisimTraceTrP∞QQ'com = bisimTraceTrP∞3 P P' PP' l (inj1 Q') tr'

bisimTraceTrP∞Q : Process ∞ {lu} c
bisimTraceTrP∞Q = lemmayyy1' bisimTraceTrP∞Qcom Q' stab'

```

bisimTraceTrP ∞ QQ'com

bisimTraceTrP ∞ tr : TrP ∞ {lu} l (inj₁ bisimTraceTrP ∞ Q) P
 bisimTraceTrP ∞ tr = lemmayyy₂' (bisimTraceTrP ∞ Qcom) l (forcep P)
Q' stab' bisimTraceTrP ∞ Q
bisimTraceTrP ∞ Q

bisimTraceTrP ∞ QQ' : Bisimw bisimTraceTrP ∞ Q Q'
 bisimTraceTrP ∞ QQ' = lemmayyy₃' bisimTraceTrP ∞ Qcom Q' stab'
bisimTraceTrP ∞ QQ'com

--@HIDE-END

bisimTraceTrP ∞ Qhat : Process ∞ {lu} c

--@HIDE-BEG

bisimTraceTrP ∞ Qhat = nonDivBecomeStable₁ c bisimTraceTrP ∞ Q
 (bisimStableImpliesNotDivergent c
 bisimTraceTrP ∞ Q Q'
 bisimTraceTrP ∞ QQ' stab'
 (stableImpliesNonDiv Q' stab'))

bisimTraceTrP ∞ trhat : TrP {lu} [] (inj₁ bisimTraceTrP ∞ Qhat) bisimTraceTrP ∞ Q
 bisimTraceTrP ∞ trhat = nonDivBecomeStable₂ c

bisimTraceTrP ∞ Q
 (bisimStableImpliesNo
 bisimTraceTrP ∞ Q
 bisimTraceTrP ∞ QQ'
 (stableImpliesNonD

--@HIDE-END

bisimTraceTrP ∞ QhatQ' : Bisimw bisimTraceTrP ∞ Qhat Q'

--@HIDE-BEG

bisimTraceTrP ∞ QhatQ' = bisimPPWithEmptyTr bisimTraceTrP ∞ Q Q' bisimTraceTrP ∞ QQ'
 (bisimStableImpliesNotDivergent c bisimTraceTrP ∞ Q Q' bisimTraceTrP ∞ QQ')

```

(stableImpliesNonDiv  $Q'$   $stab'$ )) bisimTraceTrP $\infty$ trhat

bisimTraceTrP $\infty$ stabSchQhat : stableSch bisimTraceTrP $\infty$ Qhat
bisimTraceTrP $\infty$ stabSchQhat = nonDivBecomeStable3  $c$  bisimTraceTrP $\infty$ Q
                               (bisimStableImpliesNotDivergent  $c$  bisimTraceTrP $\infty$ Q  $Q'$  bisimTraceTrP $\infty$ trhat)
                               (stableImpliesNonDiv  $Q'$   $stab'$ ))

--@HIDE-END
bisimTraceTrP $\infty$ stabQhat : stable bisimTraceTrP $\infty$ Qhat
--@HIDE-BEG
bisimTraceTrP $\infty$ stabQhat = stabSchBisim2stabRosclsStabRosc bisimTraceTrP $\infty$ Qhat  $Q'$  bisimTraceTrP $\infty$ stabSchQhat

bisimTraceTrP $\infty$ trhat1 : TrP $\infty$  { $lu$ } ( $l$  ++ []) (inj1 bisimTraceTrP $\infty$ Qhat)  $P$ 
bisimTraceTrP $\infty$ trhat1 = trPAppendTrw $\infty$   $c$   $P$  bisimTraceTrP $\infty$ Q  $l$  [] (inj1 bisimTraceTrP $\infty$ Qhat)

--@HIDE-END
bisimTraceTrP $\infty$ trhat2 : TrP $\infty$  { $lu$ }  $l$  (inj1 bisimTraceTrP $\infty$ Qhat)  $P$ 

--@END
bisimTraceTrP $\infty$ trhat2 = subst ( $\lambda l' \rightarrow$  TrP $\infty$  { $lu$ }  $l'$  (inj1 bisimTraceTrP $\infty$ Qhat)  $P$ ) eql bisimTraceTrP $\infty$ trhat1
eql : ( $l$  ++ [])  $\equiv$   $l$ 
eql = lemEqList  $l$ 

```

A.12 bisimilarityProofsWithSchneiderStable3Part2WeakerVersion

```
--@PREFIX@bisimilarityProofsWithSchneiderStablethreeParttwoWeakerVersion
```

```
module bisimilarityProofsWithSchneiderStable3Part2WeakerVersion where
```

```

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty

```

```

open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable3
open import bisimSym
open import bisimilarityProofsWithSchneiderStable

mutual
  schStabNoTraceToInj2+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
    (stabSchP : stableSch+ P)
    (a : ChoiceSet c)
    (tr : TrP+ [] (inj₂ a) P)
    (notick : ¬ (ChoiceSet (T P)))
    → ⊥

  schStabNoTraceToInj2+ P stabSchP a (intc .[] .(inj₂ a) tauP tr) = ⊥-elim (stabSchP tauP)
  schStabNoTraceToInj2+ P stabSchP .(PT P x) (terc x) y = y x

mutual

--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartOne

  stabSchBisim2stabRosclsStabRosc : {lu : LUniv}{c : Choice}
    (P P' : Process ∞ {lu} c)
    (PP' : Bisimw P P')
    (stabP' : stable P')
    (stabSchP : stableSch P)
    → stable P

  stabSchBisim2stabRosclsStabRosc
    (terminate x) (terminate x₁)

```

```

    (eqterminate terequiv) stabP' stabSchP = stabP'
stabSchBisim2stabRosclsStabRosc
  (terminate x) (node P')
  (eqterminate (termenode terequivP'))
  (P' stabSch „ notickP') stabSchP _
    = tauOrTickNoTauP'ImpliesConclusion tauOrTickNoTauP' where
      tauOrTickNoTauP' : ChoiceSet (I P') ⊔ (¬ (ChoiceSet (I P'))
        × ChoiceSet (T P'))
      tauOrTickNoTauP' = hasTauOrTickNoTau terequivP'

      tauOrTickNoTauP'ImpliesConclusion : ChoiceSet (I P')
        ⊔ (¬ (ChoiceSet (I P'))
          × ChoiceSet (T P')) → ⊥

      tauOrTickNoTauP'ImpliesConclusion (inj₁ tauChoiceP')
        = P' stabSch tauChoiceP'
      tauOrTickNoTauP'ImpliesConclusion (inj₂ (– „ tickChoiceP'))
        = notickP' tickChoiceP'

stabSchBisim2stabRosclsStabRosc
  (terminate x) .(terminate –) (eqterminater terequiv)
  stabP' stabSchP
  = stabP'
stabSchBisim2stabRosclsStabRosc
  (node P) (terminate x) PP' stabP' stabSchP
  = ⊥-elim (stabP' –)
stabSchBisim2stabRosclsStabRosc
  (node P) (node P') (eqnode bisimQQ') (stabSchP' „ noTickP')
  stabSchP
  = stabSchP „ noTickP
  where
    traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj₂ (PT P t)) P'
    traceToTickP = bisimTtr bisimQQ'

    noTickP : ¬ (ChoiceSet (T P))
    noTickP t = schStabNoTraceToInj2+ P' stabSchP' (PT P t)
      (traceToTickP t) noTickP'

```

--@END

```

stabSchBisim2stabRosclsStabRosc+ : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ P P')
  (stabP' : stable+ P')
  (stabSchP : stableSch+ P)
  → stable+ P
stabSchBisim2stabRosclsStabRosc+ P P' PP' (stabSchp' „ noTickP') stabSchP = stabSchP
  where
traceToTickP : (t : ChoiceSet (T P)) → TrP+ [] (inj2 (PT P t)) P'
traceToTickP = bisimTtr PP'

noTickP : ¬ (ChoiceSet (T P))
noTickP t = schStabNoTraceToInj2+ P' stabSchp' (PT P t) (traceToTickP t) noTickP'

```

```

stabSchBisim2stabRosclsStabRosc∞ : {lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (stabP' : stable∞ P')
  (stabSchP : stableSch∞ P)
  → stable∞ P
stabSchBisim2stabRosclsStabRosc∞ P P' PP' stabP' stabSchP = stabSchBisim2stabRosclsS
  (forcep P) (forcep P') (t

```

mutual

```

lemmaDrefusalStableNotTermequiv' : {lu : LUniv}{c : Choice}(Q : Process ∞ {lu} c)
  (stab : stable Q)
  (y : ChoiceSet c)
  (termequivQ : TerminateEquivalent y Q)
  → ⊥
lemmaDrefusalStableNotTermequiv' (terminate x) stab y termequiv = ⊥-elim (stab tt)
lemmaDrefusalStableNotTermequiv' (node Q)
  stab y (termenode terequivP) = hasTauOrTickGivesBot hasTauOrTickNoTau'
  where
    hasTauOrTickNoTau' : ChoiceSet (I Q) ⊔
      ¬ (ChoiceSet (I Q)) × ChoiceSet (T Q)
    hasTauOrTickNoTau' = hasTauOrTickNoTau terequivP

    hasTauOrTickGivesBot : ChoiceSet (I Q) ⊔ ¬ (ChoiceSet (I Q)) × ChoiceSet (T Q) → ⊥

```

```

hasTauOrTickGivesBot (inj1 x) = stabToNoInternal+ Q stab x
hasTauOrTickGivesBot (inj2 (noti „ t)) = (proj2' stab) t

```

mutual

```

lemmayyy1' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c
lemmayyy1' (inj1 Q') Q      stab x1 = Q'
lemmayyy1' (inj2 y) Q      stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q      stab y termequivQ)

```

```

lemmayyy1+' : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c
lemmayyy1+' (inj1 Q') Q stab x1 = Q'
lemmayyy1+' (inj2 y) Q      stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

```

```

lemmayyy2' : {lu : LUniv}{c : Choice} → (result :      Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (stab : stable Q)
  → (bisim : BisimForNextP result (inj1 Q))
  → TrP {lu} l result P
  → TrP {lu} l (inj1 (lemmayyy1' result Q stab bisim)) P
lemmayyy2' (inj1 Q') l P Q stab bisim tr = tr
lemmayyy2' (inj2 y) l P Q      stab termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

```

```

lemmayyy2+' : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)

```

```

→ (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab+ : stable Q)
→ (bisim : BisimForNextP result (inj1 Q))
→ TrP+ {lu} l result P
→ TrP+ {lu} l (inj1 (lemmayyy1' result Q stab+ bisim)) P
lemmayyy2+' (inj1 Q') l P Q stab bisim tr = tr
lemmayyy2+' (inj2 y) l P Q stab termequivQ x =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

lemmayyy3' : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (bisim : BisimForNextP result (inj1 Q))
→ Bisimw (lemmayyy1' result Q stab bisim) Q
lemmayyy3' (inj1 Q') Q stab bisim = bisim
lemmayyy3' (inj2 y) Q stab termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv' Q stab y termequivQ)

--\bisimilarityProofsWithSchneiderStablethreeParttwoWeakerVersion
--@BEGIN@PartTwo

module _ {lu : LUniv}{c : Choice}
(P P' : Process∞ ∞ c)
(PP' : Bisimw∞ {∞} P P')
(l : List (Label lu))
(Q' : Process ∞ {lu} c)
(tr' : TrP∞ l (inj1 Q') P') where

bisimTraceTrP∞Qcom : Process ∞ c ⊔ ChoiceSet c
bisimTraceTrP∞trcom : TrP∞ {lu} l (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') P
bisimTraceTrP∞QQ'com : BisimForNextP (bisimTraceTrP∞1 P P' PP' l
  (inj1 Q') tr') (inj1 Q')

--@END

--\bisimilarityProofsWithSchneiderStablethreeParttwo
--@BEGIN@PartTwoProof

bisimTraceTrP∞Qcom = bisimTraceTrP∞1 P P' PP' l (inj1 Q') tr'

```

```

bisimTraceTrP $\infty$ trcom = bisimTraceTrP $\infty_2$  P P' PP' l (inj1 Q') tr'
bisimTraceTrP $\infty$ QQ'com = bisimTraceTrP $\infty_3$  P P' PP' l (inj1 Q') tr'

```

```
--@END
```

A.13 bisimilarityProofsWithSchneiderStable3Part3.agda

```
module bisimilarityProofsWithSchneiderStable3Part3 where
```

```

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData
open import bisimilarityProofsWithSchneiderStable3
open import bisimilarityProofsWithSchneiderStable3Part2
open import bisimSym
open import bisimilarityProofsWithSchneiderStable

```

```

mutual
--@BEGIN@bisimRefusalros

```

```

bisimRefusalros : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P') (l : List (Label lu))
  (X : Label lu → Bool)
  (fail : failure P' l true X)
  → failure P l true X
bisimRefusalros {lu}{c} P P' PP' l X
  (stableFail (stableFp Q' tr' stab' drefuse'))
  = (stableFail (stableFp Qhat trhat₂
    stabQhat -- (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat {!s
    drefusehat))
where
  Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
  Qcom = bisimTraceTrP₁ P P' PP' l (inj₁ Q') tr'

  trcom : TrP {lu} l (bisimTraceTrP₁ P P' PP' l
    (inj₁ Q') tr') P
  trcom = bisimTraceTrP₂ P P' PP' l (inj₁ Q') tr'

  QQ'com : BisimForNextP (bisimTraceTrP₁ P P' PP' l
    (inj₁ Q') tr') (inj₁ Q')
  QQ'com = bisimTraceTrP₃ P P' PP' l (inj₁ Q') tr'

  Q : Process ∞ {lu} c
  Q = lemmayyy₁ Qcom Q' stab' X drefuse' QQ'com

  tr : TrP {lu} l (inj₁ Q) P
  tr = lemmayyy₂ Qcom l P Q' stab' X drefuse' QQ'com trcom

  QQ' : Bisimw Q Q'
  QQ' = lemmayyy₃ Qcom Q' stab' X drefuse' QQ'com

  Qhat : Process ∞ {lu} c
  Qhat = nonDivBecomeStable₁ c Q
    (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
    (stableImpliesNonDiv Q' stab'))

  trhat : TrP {lu} [] (inj₁ Qhat) Q
  trhat = nonDivBecomeStable₂ c Q
    (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
    (stableImpliesNonDiv Q' stab'))

```

```

QhatQ' :      Bisimw Qhat Q'
QhatQ' =      bisimPPWithEmptyTr Q Q' QQ' stab'
              (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
               (stableImpliesNonDiv Q' stab')) trhat

stabSchQhat : stableSch Qhat
stabSchQhat   = nonDivBecomeStable3 c Q
              (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
               (stableImpliesNonDiv Q' stab'))

stabQhat : stable Qhat
stabQhat = stabSchBisim2stabRosclsStabRosc Qhat Q' QhatQ' stab' stabSchQhat

-- stabNoTick : noTickIfRoscoe true Qhat
-- stabNoTick = {!!} -- bisimwStableToNoTick Qhat Q' QhatQ' stab' stabQhat

trhat1      : TrP {lu} (l ++ []) (inj1 Qhat) P
trhat1 = trPAppendTrw c P Q l [] (inj1 Qhat) tr trhat

eq1 : (l ++ []) ≡ l
eq1 = lemEqList l

trhat2      : TrP {lu} l (inj1 Qhat) P
trhat2      = subst (λ l' → TrP {lu} l' (inj1 Qhat) P) eq1 trhat1

drefusehat :      DRefusal Qhat true X
drefusehat =      bisimDRefusal Q' Qhat stab'
                  (BismwSym Qhat Q' QhatQ') X true drefuse'

bisimRefusalros {lu}{c} P P' PP' l X
  (divergentFailure (trdiv Q' trp' divq'))
  = (divergentFailure (trdiv Q tr divp))

where
  Qcom : Process ∞ {lu} c ⊕ ChoiceSet c
  Qcom = bisimTraceTrP1 P P' PP' l (inj1 Q') trp'

  trcom : TrP {lu} l (bisimTraceTrP1 P P' PP' l (inj1 Q') trp') P
  trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') trp'

  QQ'com : BisimForNextP

```

$$QQ'com = \text{bisimTraceTrP}_3 \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ trp' \ (\text{inj}_1 \ Q')$$

$$Q : \text{Process} \infty \{lu\} \ c$$

$$Q = \text{lemmaxx}_1 \ Qcom \ Q' \ divq' \ QQ'com$$

$$tr : \text{TrP} \ \{lu\} \ l \ (\text{inj}_1 \ Q) \ P$$

$$tr = \text{lemmaxx}_2 \ Qcom \ l \ P \ Q' \ divq' \ QQ'com \ trcom$$

$$QQ' : \text{Bisimw} \ Q \ Q'$$

$$QQ' = \text{lemmaxx}_3 \ Qcom \ l \ Q' \ divq' \ QQ'com$$

$$Q'Q : \text{Bisimw} \ Q' \ Q$$

$$Q'Q = \text{BismwSym} \ Q \ Q' \ QQ'$$

$$divp : \text{DivergentProcess} \infty \{lu\} \ c \ Q$$

$$divp = \text{bisimImpliesDivergentPreserv} \ c \ Q' \ Q \ Q'Q \ divq'$$

--@END

--@BEGIN@bisimRefusalrosplus

mutual

$$\text{bisimRefusalros+} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \ (P : \text{Process+} \infty \{lu\} \ c)$$

$$(P' : \text{Process+} \infty \{lu\} \ c)$$

$$(PP' : \text{Bisimw+} \ \{\infty\} \ P \ P')$$

$$(l : \text{List} \ (\text{Label} \ lu))$$

$$(X : \text{Label} \ lu \rightarrow \text{Bool})$$

$$(fail : \text{failure+} \ P' \ l \ \text{true} \ X)$$

$$\rightarrow \text{failure+} \ P \ l \ \text{true} \ X$$

$$\text{bisimRefusalros+} \ \{lu\} \ \{c\} \ P \ P' \ PP' \ l \ X$$

$$(\text{stableFail} \ (\text{stableFp} \ Q' \ tr' \ stab' \ drefuse'))$$

$$= (\text{stableFail} \ (\text{stableFp} \ Qhat \ trhat_2 \ stabQhat$$

$$\text{--} \ (\text{stabSchNoTickIfRos2StablePar} \ Qhat \ \text{true} \ \text{stabSchQhat} \ \text{stabNoTick})$$

$$\text{drefusehat}))$$

where

$$Qcom : \text{Process} \infty \{lu\} \ c \uplus \text{ChoiceSet} \ c$$

$$Qcom = \text{bisimTraceTrP}_1+ \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ tr'$$

```
trcom : TrP+ l (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') P
trcom =    bisimTraceTrP2+ P P' PP' l (inj1 Q') tr'
```

```
QQ'com : BisimForNextP
          (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') (inj1 Q')
QQ'com = bisimTraceTrP3+ P P' PP' l (inj1 Q') tr'
```

```
Q : Process ∞ {lu} c
Q = lemmayyy1 Qcom Q' stab' X drefuse' QQ'com
```

```
tr : TrP+ {lu} l (inj1 Q) P
tr = lemmayyy2+ Qcom l P Q' stab' X drefuse' QQ'com trcom
```

```
QQ' : Bisimw Q Q'
QQ' = lemmayyy3 Qcom Q' stab' X drefuse' QQ'com
```

```
Qhat : Process ∞ {lu} c
Qhat = nonDivBecomeStable1 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))
```

```
trhat : TrP {lu} [] (inj1 Qhat) Q
trhat = nonDivBecomeStable2 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))
```

```
QhatQ' : Bisimw Qhat Q'
QhatQ' = bisimPPWithEmptyTr Q Q' QQ' stab'
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab')) trhat
```

```
stabSchQhat : stableSch Qhat
stabSchQhat = nonDivBecomeStable3 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))
```

```
{- stabSchQhat : stableSch Qhat stabSchQhat = {!!} {-nonDivBecomeStable3 c Q (bisim
```

```
stabQhat : stable Qhat
stabQhat = stabSchBisim2stabRosclsStabRosc Qhat Q' QhatQ' stab' stabSchQhat
```

$\text{stabNoTick} : \text{noTickIfRoscoe } \text{true } \text{Qhat}$
 $\text{stabNoTick} = \text{bisimwStableToNoTick } \text{Qhat } Q' \text{QhatQ}' \text{stab}' \text{stabQhat}$

$\text{trhat}_1 : \text{TrP}+ \{lu\} (l ++ []) (\text{inj}_1 \text{Qhat}) P$
 $\text{trhat}_1 = \text{trPAppendTrw}+ c P Q l [] (\text{inj}_1 \text{Qhat}) \text{tr trhat}$

$\text{eq1} : (l ++ []) \equiv l$
 $\text{eq1} = \text{lemEqList } l$

$\text{trhat}_2 : \text{TrP}+ \{lu\} l (\text{inj}_1 \text{Qhat}) P$
 $\text{trhat}_2 = \text{subst } (\lambda l' \rightarrow \text{TrP}+ \{lu\} l' (\text{inj}_1 \text{Qhat}) P) \text{eq1 trhat}_1$

$\text{drefusehat} : \text{DRefusal } \text{Qhat } \text{true } X$
 $\text{drefusehat} = \text{bisimDRefusal } Q' \text{Qhat } \text{stab}'$
 $(\text{BismwSym } \text{Qhat } Q' \text{QhatQ}') X \text{true } \text{drefuse}'$

$\text{bisimRefusalros}+ \{lu\}\{c\} P P' PP' l X$
 $(\text{divergentFailure } (\text{trdiv } Q' \text{trp}' \text{divq}'))$
 $= (\text{divergentFailure } (\text{trdiv } Q \text{tr divp}))$

where

$\text{Qcom} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1+ \{lu\} P P' PP' l (\text{inj}_1 Q') \text{trp}'$

$\text{trcom} : \text{TrP}+ \{lu\} l (\text{bisimTraceTrP}_1+ P P' PP' l (\text{inj}_1 Q') \text{trp}') P$
 $\text{trcom} = \text{bisimTraceTrP}_2+ P P' PP' l (\text{inj}_1 Q') \text{trp}'$

$\text{QQ'com} : \text{BisimForNextP}$
 $(\text{bisimTraceTrP}_1+ P P' PP' l (\text{inj}_1 Q') \text{trp}') (\text{inj}_1 Q')$
 $\text{QQ'com} = \text{bisimTraceTrP}_3+ P P' PP' l (\text{inj}_1 Q') \text{trp}'$

$Q : \text{Process } \infty \{lu\} c$
 $Q = \text{lemmaxxx}_1 \text{Qcom } Q' \text{divq}' \text{QQ'com}$

$\text{tr} : \text{TrP}+ \{lu\} l (\text{inj}_1 Q) P$
 $\text{tr} = \text{lemmaxxx}_2+ \text{Qcom } l P Q' \text{divq}' \text{QQ'com } \text{trcom}$

$\text{QQ}' : \text{Bisimw } Q Q'$
 $\text{QQ}' = \text{lemmaxxx}_3 \text{Qcom } l Q' \text{divq}' \text{QQ'com}$

```

Q'Q : Bisimw Q' Q
Q'Q = BismwSym Q Q' QQ'

divp : DivergentProcess ∞ {lu} c Q
divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divq'

--@END

--@BEGIN@bisimImFdiTwo

bisimImFDl2 : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
              (P' : Process ∞ {lu} c)
              (PP' : Bisimw {∞} P P')
              → P ⊑fdi2ros P'
bisimImFDl2 {lu}{c} P P' PP' = bisimRefusalros P P' PP'

bisimImFDl2r : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
              (P' : Process ∞ {lu} c)
              (PP' : Bisimw {∞} P P')
              → P' ⊑fdi2ros P
bisimImFDl2r {lu}{c} P P' PP' = bisimImFDl2 P' P (BismwSym P P' PP')

--@END

bisimImFDl2+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
              (P' : Process+ ∞ {lu} c)
              (PP' : Bisimw+ {∞} P P')
              → P ⊑fdi2ros+ P'
bisimImFDl2+ {lu}{c} P P' PP' = bisimRefusalros+ P P' PP'

bisimImFDl2r+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
              (P' : Process+ ∞ {lu} c)
              (PP' : Bisimw+ {∞} P P')
              → P' ⊑fdi2ros+ P
bisimImFDl2r+ {lu}{c} P P' PP' = bisimImFDl2+ P' P (BismwSym+ P P' PP')

```

A.14 bisimilarityProofsWithSchneiderStable.agda

```
--@PREFIX@bisimilarityProofs

module bisimilarityProofsWithSchneiderStable where

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool      renaming (T to True)
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import fdi
open import auxData

mutual
  nondivImpliesIPorNotIPS : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
    (nondiv : NonDivergent+ P)
    → ChoiceSet (I P) ⊔ ¬ (ChoiceSet (I P))
  nondivImpliesIPorNotIPS {c} P
    (nondiv - chemptyornot) = chemptyornot

mutual
```

```

swapChoiceSetsS : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (nondiv : NonDivergent+ P)
  → ChoiceSet (I P') ⊔ ¬ (ChoiceSet (I P'))
swapChoiceSetsS {c} P P' PP' nond
  = nondivImpliesIPnotIPS P' (nondiv+ PP' nond)

```

mutual

```

stableImpliesNonDiv∞S : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (PS : stable∞ P) → NonDivergent∞ P
forceND (stableImpliesNonDiv∞S P PS) = stableImpliesNonDivS (forceP P) PS

stableImpliesNonDivS : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (PS : stable P)
  → NonDivergent P
stableImpliesNonDivS (terminate x) PS = tt
stableImpliesNonDivS (node x) PS = stableImpliesNonDiv+S x PS

stableImpliesNonDiv+S : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (PS : stable+ P) → NonDivergent+ P
stableImpliesNonDiv+S P PS = nondiv
  (stableImpliesNonDiv+auxS P PS) (inj2 (stabToNoInternal+ P PS))

stableImpliesNonDiv+auxS : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (PS : stable+ P)(i : ChoiceSet (I P))
  → NonDivergent∞ (PI P i)
stableImpliesNonDiv+auxS P PS i = ⊥-elim (stabToNoInternal+ P PS i)

```

mutual

```

TerImpliesNotDivergentaux+S : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent+ a P)
  → NonDivergent+ P
TerImpliesNotDivergentaux+S c P a terequiv = nondiv
  (λ i → TerImpliesNotDivergentaux∞S c
    (PI P i) a (onlyIntChoice terequiv i))
  (hasTauOrNotTau terequiv)

```

```

TerImpliesNotDivergentauxS : {lu : LUniv}(c : Choice)(P : Process ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent a P)
  → NonDivergent P
TerImpliesNotDivergentauxS c (terminate x) a terequiv = tt
TerImpliesNotDivergentauxS c (node x) a (termeqnode terequivP)
  = TerImpliesNotDivergentaux+S c x a terequivP

TerImpliesNotDivergentaux∞S : {lu : LUniv}(c : Choice)(P : Process∞ ∞ {lu} c)
  (a : ChoiceSet c)
  (terequiv : TerminateEquivalent a (forceP P))
  → NonDivergent∞ P
forceND (TerImpliesNotDivergentaux∞S c P a terequiv)
  = TerImpliesNotDivergentauxS c (forceP P) a terequiv

```

```
--@BEGIN@bisimStableImpliesNotDivergent
```

```
mutual
```

```

bisimStableImpliesNotDivergent∞S : {lu : LUniv}(c : Choice)
  (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (PS' : stable∞ P')
  (nonDivP' : NonDivergent∞ P')
  → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞S c P P' PP' PS' nonDivP')
  = bisimStableImpliesNotDivergentS c (forceP P)
    (forceP P')
    (forceB PP')
    PS' (forceND nonDivP')

bisimStableImpliesNotDivergentS : {lu : LUniv}(c : Choice)
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (PS' : stable P')
  (nonDivP' : NonDivergent P')
  → NonDivergent P
bisimStableImpliesNotDivergentS c (terminate x) P' PP' PS' nonDivP' = tt
bisimStableImpliesNotDivergentS c (node P) (terminate a)

```

```

      (eqterminater (termeqnode terequivP))
      PS' nonDivP' =
      TerImpliesNotDivergentauxS c (node P)
      a ((termeqnode terequivP))

bisimStableImpliesNotDivergentS c (node P) (node P') (eqnode PP') PS'
nonDivP' = bisimStableImpliesNotDivergent+S
      c P P' PP' PS' nonDivP'

bisimStableImpliesNotDivergent+S : {lu : LUniv}{c : Choice}
      (P P' : Process+ ∞ {lu} c)
      (PP' : Bisimw+ P P')
      (PS' : stable+ P')
      (nonDivP' : NonDivergent+ P')
      → NonDivergent+ P
bisimStableImpliesNotDivergent+S c P P' PP' PS' nonDivP'
      = nondiv+r PP' nonDivP'

--@END

--@BEGIN@nonDivBecomeStable

mutual
nonDivBecomeStable∞1S : {lu : LUniv}{c : Choice}
      (P : Process∞ ∞ {lu} c)
      (nonDivP : NonDivergent∞ P)
      → Process ∞ {lu} c
nonDivBecomeStable∞1S c P nonDivP = nonDivBecomeStable1S
      c (forcep P) (forceND nonDivP)

nonDivBecomeStable∞2S : {lu : LUniv}{c : Choice}
      (P : Process∞ ∞ {lu} c)
      (nonDivP : NonDivergent∞ P)
      → TrP∞ {lu} [](inj1
      (nonDivBecomeStable∞1S c P nonDivP)) P
nonDivBecomeStable∞2S c P nonDivP = nonDivBecomeStable2S
      c (forcep P) (forceND nonDivP)

nonDivBecomeStable∞3S : {lu : LUniv}{c : Choice}
      (P : Process∞ ∞ {lu} c)

```

$$\begin{aligned}
& (nonDivP : \text{NonDivergent} \infty P) \\
& \rightarrow \text{stableSch} (\text{nonDivBecomeStable} \infty_1 S \\
& \quad c P nonDivP) \\
\text{nonDivBecomeStable} \infty_3 S \ c \ P \ nonDivP &= \text{nonDivBecomeStable} \infty_3 S \\
& \quad c (\text{forceP} P) (\text{forceND} nonDivP)
\end{aligned}$$

$$\begin{aligned}
\text{nonDivBecomeStable} +_1 S : \{lu : LUniv\} (c : \text{Choice}) \\
& (P : \text{Process} + \infty \{lu\} c) \\
& (nonDivP : \text{NonDivergent} + P) \\
& \rightarrow \text{Process} \infty \{lu\} c \\
\text{nonDivBecomeStable} +_1 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_1 \ int)) &= \\
& \quad \text{nonDivBecomeStable} \infty_1 S \\
& \quad c (\text{PI} P int) (x int) \\
\text{nonDivBecomeStable} +_1 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_2 \ stab)) &= \text{node} P
\end{aligned}$$

$$\begin{aligned}
\text{nonDivBecomeStable} +_2 S : \{lu : LUniv\} (c : \text{Choice}) \\
& (P : \text{Process} + \infty \{lu\} c) \\
& (nonDivP : \text{NonDivergent} + P) \\
& \rightarrow \text{TrP} + \{lu\} [] (\text{inj}_1 \\
& \quad (\text{nonDivBecomeStable} +_1 S \ c \ P \ nonDivP)) P \\
\text{nonDivBecomeStable} +_2 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_1 \ int)) &= \text{intc} [] (\text{inj}_1 \\
& \quad (\text{nonDivBecomeStable} +_1 S \ c \ P \\
& \quad (\text{nondiv} \ x \ (\text{inj}_1 \ int)))) int \\
& \quad (\text{nonDivBecomeStable} \infty_2 S \ c \\
& \quad (\text{PI} P int) (x int)) \\
\text{nonDivBecomeStable} +_2 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_2 \ stab)) &= \text{empty}
\end{aligned}$$

$$\begin{aligned}
\text{nonDivBecomeStable} +_3 S : \{lu : LUniv\} (c : \text{Choice}) \\
& (P : \text{Process} + \infty \{lu\} c) \\
& (nonDivP : \text{NonDivergent} + P) \\
& \rightarrow \text{stableSch} \text{--} \text{stable} \\
& \quad (\text{nonDivBecomeStable} +_1 S \ c \ P \ nonDivP) \\
\text{nonDivBecomeStable} +_3 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_1 \ int)) &= \\
& \quad \text{nonDivBecomeStable} \infty_3 S \ c \\
& \quad (\text{PI} P int) (x int) \\
\text{nonDivBecomeStable} +_3 S \ c \ P \ (\text{nondiv} \ x \ (\text{inj}_2 \ stab)) &= \text{stab}
\end{aligned}$$

$$\begin{aligned}
\text{nonDivBecomeStable}_1 S : \{lu : LUniv\} (c : \text{Choice}) \\
& (P : \text{Process} \infty \{lu\} c) \\
& (nonDivP : \text{NonDivergent} P)
\end{aligned}$$

```

→ Process ∞ {lu} c
nonDivBecomeStable1S c (terminate x) nonDivP = terminate x
nonDivBecomeStable1S c (node x) nonDivP =
    nonDivBecomeStable+1S c x nonDivP

nonDivBecomeStable2S : {lu : LUniv}(c : Choice)
    (P : Process ∞ {lu} c)
    (nonDivP : NonDivergent P)
→ TrP {lu} [] (inj1
    (nonDivBecomeStable1S c P nonDivP)) P
nonDivBecomeStable2S c (terminate x) nonDivP = empty x
nonDivBecomeStable2S c (node x) nonDivP
    = tnode (nonDivBecomeStable+2S c x nonDivP)

nonDivBecomeStable3S : {lu : LUniv}(c : Choice)
    (P : Process ∞ {lu} c)
    (nonDivP : NonDivergent P)
→ stableSch (nonDivBecomeStable1S c P nonDivP)
nonDivBecomeStable3S c (terminate x) nonDivP = _
nonDivBecomeStable3S c (node x) nonDivP = nonDivBecomeStable+3S c
    x nonDivP

```

--@END

mutual

```

nonDivBecomesStableBisimProof∞S : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
    (nondiv' : NonDivergent∞ P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent∞ a P)
→ Bisimw (nonDivBecomeStable∞1S c P nondiv') (terminate a)
nonDivBecomesStableBisimProof∞S P nondiv' a terequivP =
    nonDivBecomesStableBisimProofS (forceP P) (forceND nondiv') a terequivP

nonDivBecomesStableBisimProofS : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
    (nondiv' : NonDivergent P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent a P)
→ Bisimw (nonDivBecomeStable1S c P nondiv') (terminate a)

```

```

nonDivBecomesStableBisimProofS (terminate x) nondiv' .x termeqterm
    = BismwRef (terminate x)
nonDivBecomesStableBisimProofS (node x) nondiv' a
    (termeqnode terequivP) =
    nonDivBecomesStableBisimProof+S x nondiv' a terequivP

```

```

nonDivBecomesStableBisimProof+S : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
    (nondiv' : NonDivergent+ P)
    (a : ChoiceSet c)
    (terequivP : TerminateEquivalent+ a P)
    → Bisimw (nonDivBecomeStable+_1S c P nondiv') (terminate a)
nonDivBecomesStableBisimProof+S P (nondiv x (inj1 int)) a terequivP
    = nonDivBecomesStableBisimProof∞S (PI P int) (x int) a
    (onlyIntChoice terequivP int)
nonDivBecomesStableBisimProof+S P (nondiv x (inj2 stab)) a terequivP
    = eqterminater (termeqnode terequivP)

```

mutual

```

emptyTrPtoQImpliesEqS : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
    (PS' : stable P)(tr : TrP {lu} [] (inj1 P') P)
    → P ≡ P'
emptyTrPtoQImpliesEqS (terminate x) .(terminate x) pat (empty .x) = refl
emptyTrPtoQImpliesEqS (node x) .(node x) PS' (tnode empty) = refl
emptyTrPtoQImpliesEqS (node Q) P' PS'
    (tnode (intc .[] .(inj1 P') x1 x2)) = ⊥-elim (stabToNoInternal+ Q PS' x1)
emptyTrPtoQImpliesEq+S : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(P' : Process+ ∞ {lu} c)
    (PS' : stable+ P)(tr : TrP+ {lu} [] (inj1 P') P)
    → node P ≡ P'
emptyTrPtoQImpliesEq+S P P' PS' tr
    = emptyTrPtoQImpliesEqS (node P) P' PS' (tnode tr)

```

--@BEGIN@bisimPPWithEmptyTr

mutual

```

bisimPPWithEmptyTr∞S : {lu : LUniv}{c : Choice}
    (P P' : Process∞ ∞ {lu} c)

```

```

      (PP' : Bisimw $\infty$  P P') (PS' : stable $\infty$  P')
      (nonDivP : NonDivergent $\infty$  P)
      (tr : TrP $\infty$  {lu} [])
      (inj1 (nonDivBecomeStable $\infty_1$ S c P nonDivP)) P)
    → Bisimw (nonDivBecomeStable $\infty_1$ S c P nonDivP)
      (forceP P')
bisimPPWithEmptyTr $\infty$ S P P' PP' PS' nonDivP tr =
    bisimPPWithEmptyTrS (forceP P) (forceP P')
      (forceB PP') PS' (forceND nonDivP) tr

bisimPPWithEmptyTrS : {lu : LUniv}{c : Choice}
  (P P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw P P') (PS' : stable P')
  (nonDivP : NonDivergent P)
  (tr : TrP {lu} [] (inj1
    (nonDivBecomeStable1S {lu} c P nonDivP)) P)
  → Bisimw (nonDivBecomeStable1S {lu} c P nonDivP) P'
bisimPPWithEmptyTrS {lu} {c} .(terminate x) (terminate x1)
  PP' PS' nonDivP (empty x) = PP'
bisimPPWithEmptyTrS (node P) (terminate a)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv nondivPI (inj1 x)) (tnode tr) =
  nonDivBecomesStableBisimProof $\infty$ S (PI P x)
  (nondivPI x) a (onlyIntChoice terequivP x)
bisimPPWithEmptyTrS (node P) (terminate x)
  (eqterminater (termeqnode terequivP))
  PS' (nondiv x1 (inj2 y)) (tnode tr) =
  eqterminater (termeqnode terequivP)
bisimPPWithEmptyTrS (terminate P) (node P') PP' PS' nonDivP tr =
  PP'
bisimPPWithEmptyTrS (node P) (node P') (eqnode bisimPP') PS'
  (nondiv x chemptyornot) (tnode tr) =
  bisimPPWithEmptyTr+S P P' bisimPP'
  PS' (nondiv x chemptyornot) tr

bisimPPWithEmptyTr+S : {lu : LUniv}{c : Choice}
  (P P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ P P') (PS' : stable+ P')

```

```

      (nonDivP : NonDivergent+ P)
      (tr : TrP+ {lu} [] (inj1
        (nonDivBecomeStable+1S c P nonDivP)) P)
    → Bisimw (nonDivBecomeStable+1S c P nonDivP)
      (node P')
bisimPPWithEmptyTr+S {lu}{c} P P' PP' PS'
  (nondiv nondiv' (inj1 x1)) tr = PP'''

```

where

```

P'~ : Process∞ ∞ {lu} c
P'~ = bisimIP' PP' x1

trP'P'~ : P' →+*[ [] ] (forcep (P'~))
trP'P'~ = bisimltr PP' x1

```

```

P'≡P'~ : node P' ≡ forcep P'~ {∞}
P'≡P'~ = emptyTrPtoQImpliesEq+S P'
      (forcep P'~) PS' trP'P'~

```

```

P'~≡P' : forcep P'~ {∞} ≡ node P'
P'~≡P' rewrite P'≡P'~ = refl

```

```

P'~stable : stable (forcep P'~)
P'~stable rewrite P'~≡P' = PS'

```

```

PP'' : Bisimw (nonDivBecomeStable1S c
  (forcep (PI P x1)) (forceND (nondiv' x1)) )
  (forcep P'~)
PP'' = bisimPPWithEmptyTr+S (forcep (PI P x1))
  (forcep P'~ {∞})
  (forceB (bisimlnext PP' x1))
  P'~stable (forceND (nondiv' x1))
  (nonDivBecomeStable2S c
    (forcep (PI P x1)) (forceND (nondiv' x1)))

```

```

PP''' : Bisimw (nonDivBecomeStable∞1S c
  (PI P x1) (nondiv' x1)) (node P')
PP''' rewrite P'≡P'~ = PP''

```

```

bisimPPWithEmptyTr+S P P' PP' PS'
  (nondiv x (inj2 y)) empty = eqnode PP'

```

```

bisimPPWithEmptyTr+S P P' PP' PS'
  (nondiv x (inj2 y))
  (intc .[] .(inj1
    (node P)) x1 x2) = eqnode PP'

```

```
--@END
```

```
mutual
```

```

choicesetornotBismS : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)(PP' : Bisims+
  (cc' : ChoiceSet (I P) ⊔ ⊔ (ChoiceSet (I P)))
  → ChoiceSet (I P') ⊔ ⊔ (ChoiceSet (I P')))

```

```

choicesetornotBismS {c} P P' PP' (inj1 ip) = inj1 (bisim2l PP' ip)
choicesetornotBismS {c} P P' PP' (inj2 notip) = inj2 (λ ip' → notip (bisim2lr PP' ip'))

```

```
mutual
```

```

nondivLemBisims∞S : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  → Bisims∞ {i} P P'
  → NonDivergent∞ {i} P → NonDivergent∞ {i} P'
forceND (nondivLemBisims∞S P P' PP' nP) = nondivLemBisimsS (forceP P) (forceP P') (forceB PP' nP)

nondivLemBisimsS : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → Bisims {i} P P'
  → NonDivergent {i} P → NonDivergent {i} P'
nondivLemBisimsS .(terminate a) .(terminate a) (eqterminate {a}) nP = tt
nondivLemBisimsS .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') nP = nondivLemBisims+S Q Q' Q

```

```

nondivLemBisims+S : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  → Bisims+ {i} P P'
  → NonDivergent+ {i} P → NonDivergent+ {i} P'
nondivLemBisims+S P P' PP' (nondiv f p) =
  nondiv (λ i → nondivLemBisims∞S (PI P (bisim2lr PP' i)) (PI P' i) (bisimlNexttr PP' i)
    (f (bisim2lr PP' i)) ) (choicesetornotBismS P P' PP' p)

```

mutual

$\text{divLemBisims}\infty\text{S} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process}\infty \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims}\infty \{i\}\ P\ P'$
 $\rightarrow \text{DivergentProcess}\infty\ i\ c\ P \rightarrow \text{DivergentProcess}\infty\ i\ c\ P'$
 $\text{divLemBisims}\infty\text{S}\ P\ P'\ PP'\ nP.\text{forcediv} = \text{divLemBisimsS}\ (\text{forcep}\ P)\ (\text{forcep}\ P')\ (\text{forceB}\ PP'$
 $\text{divLemBisimsS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process} \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims} \{i\}\ P\ P'$
 $\rightarrow \text{DivergentProcess}\ i\ c\ P \rightarrow \text{DivergentProcess}\ i\ c\ P'$
 $\text{divLemBisimsS}.\text{(terminate}\ a)\ .\text{(terminate}\ a)\ (\text{eqterminate}\ \{a\})\ nP = nP$
 $\text{divLemBisimsS}.\text{(node}\ P)\ .\text{(node}\ P')\ (\text{eqnode}\ \{P\}\ \{P'\}\ PP')\ (\text{div}\ P\ \text{divP}) = \text{div}\ P'\ (\text{divLemBisimsS}.\text{(node}\ P')\ .\text{(node}\ P)\ (\text{eqnode}\ \{P'\}\ \{P\}\ PP'))$

$\text{divLemBisims}+\text{S} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process}+ \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims}+ \{i\}\ P\ P'$
 $\rightarrow \text{DivergentProcess}+ \ i\ c\ P \rightarrow \text{DivergentProcess}+ \ i\ c\ P'$
 $\text{divLemBisims}+\text{S}\ P\ P'\ PP'\ (\text{div}+ \text{int}\ Q) = \text{div}+ (\text{bisim2I}\ PP'\ \text{int})$
 $(\text{divLemBisims}\infty\text{S}\ (\text{PI}\ P\ \text{int})\ (\text{PI}\ P'\ (\text{bisim2I}\ PP'\ \text{int})))\ (\text{bisimINext}\ PP'\ \text{int})\ Q)$

mutual

$\text{nondivLemBisims}\infty\text{rS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process}\infty \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims}\infty \{i\}\ P\ P'$
 $\rightarrow \text{NonDivergent}\infty \{i\}\ P' \rightarrow \text{NonDivergent}\infty \{i\}\ P$
 $\text{forceND}\ (\text{nondivLemBisims}\infty\text{rS}\ P\ P'\ PP'\ nP) = \text{nondivLemBisimsrS}\ (\text{forcep}\ P)\ (\text{forcep}\ P')$
 $\text{nondivLemBisimsrS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process} \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims} \{i\}\ P\ P'$
 $\rightarrow \text{NonDivergent} \{i\}\ P' \rightarrow \text{NonDivergent} \{i\}\ P$
 $\text{nondivLemBisimsrS}.\text{(terminate}\ a)\ .\text{(terminate}\ a)\ (\text{eqterminate}\ \{a\})\ nP = \text{tt}$
 $\text{nondivLemBisimsrS}.\text{(node}\ Q)\ .\text{(node}\ Q')\ (\text{eqnode}\ \{Q\}\ \{Q'\}\ QQ')\ nP = \text{nondivLemBisims}+\text{rS}.\text{(node}\ Q)\ .\text{(node}\ Q')\ (\text{eqnode}\ \{Q\}\ \{Q'\}\ QQ'))$

$\text{nondivLemBisims}+\text{rS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P\ P' : \text{Process}+ \infty \{lu\}\ c)$
 $\rightarrow \text{Bisims}+ \{i\}\ P\ P'$
 $\rightarrow \text{NonDivergent}+ \{i\}\ P' \rightarrow \text{NonDivergent}+ \{i\}\ P$
 $\text{nondivLemBisims}+\text{rS}\ P\ P'\ PP'\ (\text{nondiv}\ f\ p) =$
 $\text{nondiv}\ (\lambda\ i \rightarrow \text{nondivLemBisims}\infty\text{rS}\ (\text{PI}\ P\ i)\ ((\text{PI}\ P'\ (\text{bisim2I}\ PP'\ i)))\ (\text{bisimINext}\ PP'\ i))$
 $(f\ (\text{bisim2I}\ PP'\ i))\ (\text{swapChoiceSetssr}\ P\ P'\ PP'\ p)$

mutual

$\text{divLemBisims}\infty\text{rS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process}\infty \infty \{lu\} c)$
 $\rightarrow \text{Bisims}\infty \{i\} P P'$
 $\rightarrow \text{DivergentProcess}\infty i c P' \rightarrow \text{DivergentProcess}\infty i c P$

$\text{divLemBisims}\infty\text{rS} \{i\} P P' PP' nP .\text{forcediv} = \text{divLemBisimsrS} (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP') ($

$\text{divLemBisimsrS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process} \infty \{lu\} c)$
 $\rightarrow \text{Bisims} \{i\} P P'$
 $\rightarrow \text{DivergentProcess} i c P' \rightarrow \text{DivergentProcess} i c P$

$\text{divLemBisimsrS} .(\text{terminate } a) .(\text{terminate } a) (\text{eqterminate } \{a\}) nP = nP$
 $\text{divLemBisimsrS} .(\text{node } P') .(\text{node } P) (\text{eqnode } \{P\} \{P\} PP') (\text{div } P \text{ div } P) = \text{div } P' (\text{divLemBisimsrS} .$

$\text{divLemBisims+rS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process+} \infty \{lu\} c)$
 $\rightarrow \text{Bisims+} \{i\} P P'$
 $\rightarrow \text{DivergentProcess+} i c P' \rightarrow \text{DivergentProcess+} i c P$

$\text{divLemBisims+rS} P P' PP' (\text{div+ } \text{int } Q) = \text{div+} ((\text{bisim2lr } PP' \text{ int}))$
 $((\text{divLemBisims}\infty\text{rS} (\text{PI } P ((\text{bisim2lr } PP' \text{ int})))) (\text{PI } P' \text{ int}) ((\text{bisim2lr } PP' \text{ int})))$

mutual

$\text{stabLemBisims}\infty\text{S} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process}\infty \infty \{lu\} c)$
 $\rightarrow \text{Bisims}\infty P P'$
 $\rightarrow \text{stable}\infty P' \rightarrow \text{stable}\infty P$

$\text{stabLemBisims}\infty\text{S} P P' PP' PS' = \text{stabLemBisimsS} (\text{forcep } P) (\text{forcep } P') (\text{forceB } PP') PS'$

$\text{stabLemBisimsS} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process} \infty \{lu\} c)$
 $\rightarrow \text{Bisims} P P'$
 $\rightarrow \text{stable} P' \rightarrow \text{stable} P$

$\text{stabLemBisimsS} .(\text{terminate } a) .(\text{terminate } a) (\text{eqterminate } \{a\}) PS' = PS'$

$\text{stabLemBisimsS} .(\text{node } P) .(\text{node } P') (\text{eqnode } \{P\} \{P\} PP') PS' = \text{stabLemBisims+S} P P' PP'$

$\text{stabLemBisims+S} : \{i : \text{Size}\}\{lu : \text{LUniv}\}\{c : \text{Choice}\}(P P' : \text{Process+} \infty \{lu\} c)$
 $\rightarrow \text{Bisims+} \{i\} P P'$
 $\rightarrow \text{stable+} P' \rightarrow \text{stable+} P$

$\text{stabLemBisims+S} P P' PP' (PnoI _, PNoTick) = (\lambda \text{int} \rightarrow PnoI (\text{bisim2l } PP' \text{ int})) _, (\lambda t \rightarrow PNoTick t)$

mutual

divergentImpliesNotTermEquiv+S : {lu : LUniv}(c : Choice)
 (P : Process+ ∞ {lu} c)
 (a : ChoiceSet c)
 (terP : TerminateEquivalent+ a P)
 (divP : DivergentProcess+ ∞ {lu} c P)
 → ⊥

divergentImpliesNotTermEquiv+S c P a terequivP (div+ int divP) =
 divergentImpliesNotTermEquiv∞S c (PI P int) a (onlyIntChoice terequivP)

divergentImpliesNotTermEquiv∞S : {lu : LUniv}(c : Choice)
 (P : Process∞ ∞ {lu} c)
 (a : ChoiceSet c)
 (terP : TerminateEquivalent∞ a P)
 (divP : DivergentProcess∞ ∞ {lu} c P)
 → ⊥

divergentImpliesNotTermEquiv∞S c P a terP divP = divergentImpliesNotTermEquivS c (forceP)

divergentImpliesNotTermEquivS : {lu : LUniv}(c : Choice)
 (P : Process ∞ {lu} c)
 (a : ChoiceSet c)
 (terP : TerminateEquivalent a P)
 (divP : DivergentProcess ∞ {lu} c P)
 → ⊥

divergentImpliesNotTermEquivS c .(node P) a (termegnode terequivP) (div P divP) =
 divergentImpliesNotTermEquiv+S c P

mutual

bisimImpliesDivergentPreserv∞S : {lu : LUniv}(c : Choice) (P P' : Process∞ ∞ {lu} c)
 (PP' : Bisimw∞ {∞} P P')
 (divP : DivergentProcess∞ ∞ {lu} c P)
 → DivergentProcess∞ ∞ {lu} c P'
 forcediv (bisimImpliesDivergentPreserv∞S c P P' PP' divP) =
 bisimImpliesDivergentPreservS c (forceP) (forceP') (forceB PP')

(forcediv divP)

```

bisimImpliesDivergentPreserv+S : {lu : LUniv}(c : Choice) (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  (divP : DivergentProcess+ ∞ {lu} c P)
  → DivergentProcess+ ∞ {lu} c P'
bisimImpliesDivergentPreserv+S c P P' PP' divP = bisimdiv PP' divP

```

--@BEGIN@bisimImpliesDivergentPreserv

```

bisimImpliesDivergentPreservS :
  {lu : LUniv}(c : Choice)
  (P P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (divP : DivergentProcess ∞ {lu} c P)
  → DivergentProcess ∞ {lu} c P'
bisimImpliesDivergentPreservS c .(terminate _) P' (eqterminate x) ()
bisimImpliesDivergentPreservS c .(node P) .(terminate a) (eqterminater {a}
  {.(node P)} (termeqnode terequivP)) (div P divP)
  = ⊥-elim (divergentImpliesNotTermEquiv+S c P a terequivP divP)
bisimImpliesDivergentPreservS c .(node P) .(node P') (eqnode {P} {P'} PP')
  (div P divP)
  = div P' (bisimImpliesDivergentPreserv+S c P P' PP' divP)

```

--@END

mutual

```

bisimStableImpliesNotDivergent∞'S : {lu : LUniv}(c : Choice) (P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ P P')
  (PS' : stable∞ P')
  → NonDivergent∞ P
forceND (bisimStableImpliesNotDivergent∞'S c P P' PP' PS') =
  bisimStableImpliesNotDivergent'S c (forceP P)
  (forceP P') (forceB PP') PS'

```

```

bisimStableImpliesNotDivergent'S : {lu : LUniv}(c : Choice) (P P' : Process ∞ {lu} c)
  (PP' : Bisimw P P')
  (PS' : stable P')
  → NonDivergent P

```

```

bisimStableImpliesNotDivergent'S c (terminate x) P' PP' PS' = tt
bisimStableImpliesNotDivergent'S c (node P) .(terminate a)
    (eqterminater {a} terequiv) PS' =
    TerImpliesNotDivergentauxS c (node P) a terequiv
bisimStableImpliesNotDivergent'S c (node P) .(node P')
    (eqnode {.P} {P'} PP') PS' =
    bisimStableImpliesNotDivergent+'S c P P' PP' PS'

bisimStableImpliesNotDivergent+'S : {lu : LUniv}(c : Choice) (P P' : Process+ ∞ {lu} c)
    (PP' : Bisimw+ P P') (PS' : stable+ P')
    → NonDivergent+ P
bisimStableImpliesNotDivergent+'S c P P' PP' PS' =
    nondiv+r PP' (nondiv (λ i → ⊥-elim (stabToNoInternal+ P' PS' i)))

```

```

lemBisimDRefusalAuxS : {lu : LUniv}(c : Choice)
    (x : ChoiceSet c)
    (P : Process+ ∞ {lu} c)
    (hasTauorTickNoTau : ChoiceSet (I P) ⊔ (¬ (ChoiceSet (I P)))
    (PS : ChoiceSet (I P) → ⊥)
    → ChoiceSet (T P)
lemBisimDRefusalAuxS c x P (inj1 x1) PS = ⊥-elim (PS x1)
lemBisimDRefusalAuxS c x P (inj2 (– „ x1)) PS = x1

```

mutual

```

bisimDRefusal+S : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c) (P' : Process+ ∞ {lu} c)
    (PS : stable+ P)
    (PP' : Bisimw+ {∞} P P')
    (X : Label lu → Bool)
    (isRoscoe : Bool)
    (ref : DRefusal+ P isRoscoe X)
    → DRefusal+ P' isRoscoe X
bisimDRefusal+S {c} P P' PS PP' X isRoscoe (drefusal noextChInX noTerm) =
    drefusal (bisimDRefusal+NoExtChInXS P P' PS PP' X noextChInX)
    (bisimDRefusalNoTicksIfIsRoscoeS P P' PS PP' isRoscoe noTerm)

```

```

bisimDRefusalNoTicksIfIsRoscoeS : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)

```

```

(P' : Process+ ∞ {lu} c)
(PS : stable+ P)
(PP' : Bisimw+ {∞} P P')
(isRoscoe : Bool)
(ref : NoTicksIfsRoscoe P isRoscoe)
→ NoTicksIfsRoscoe P' isRoscoe
bisimDRefusalNoTicksIfsRoscoeS {lu} {c} P P' PS      PP' isRoscoe ref tickIsIncl x =
                                                         ref tickIsIncl (lemS c P (PT P' x) path PS)
where
path : TrP+ {lu} [] (inj2 (PT P' x)) P
path = bisimTtrr PP' x

```

```

lemS : {lu : LUniv} (c : Choice) (P : Process+ ∞ {lu} c) (x : ChoiceSet c)
      (tr : TrP+ {lu} [] (inj2 x) P)
      (PS : stable+ P) → ChoiceSet (T P)
lemS c P x (intc .[] .(inj2 x) x' x2) PS = ⊥-elim (stabToNoInternal+ P PS x')
lemS c P .(PT P x1) (terc x1) PS = x1

```

```

bisimDRefusal+NoExtChInXS : {lu : LUniv} {c : Choice} (P : Process+ ∞ {lu} c)
(P' : Process+ ∞ {lu} c)
(PS : stable+ P)
(PP' : Bisimw+ {∞} P P')
(X : Label lu → Bool)
(ref : NoExtChInX P X)
→ NoExtChInX P' X
bisimDRefusal+NoExtChInXS {lu} {c} P P' PS      PP' X ref e x = lem2S c P Q (Lab P' e) tr PS X
where
Q : Process ∞ {lu} c
Q = forcep (PP' .bisimEP'r e)
tr : TrP+ {lu} (Lab P' e :: []) (inj1 Q) P
tr = PP' .bisimEtrr e

```

```

lem2S : {lu : LUniv} (c : Choice) (P : Process+ ∞ {lu} c) (Q : Process ∞ {lu} c)
      (l : Label lu) (tr : TrP+ {lu} (l :: []) (inj1 Q) P)
      (PS : stable+ P) (X : Label lu → Bool) (ref : NoExtChInX P X) (labX : True (X l)) → ⊥
lem2S c P Q .(Lab P x) (extc .[] .(inj1 Q) x x1) PS X ref labX = ref x labX
lem2S c P Q l (intc .(l :: []) .(inj1 Q) x x1) PS X ref labX = stabToNoInternal+ P PS x

```

```

bisimDRefusalS : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c) (P' : Process ∞ {lu} c)
  (PS : stable P)
  (PP' : Bisimw {∞} P P')
  (X : Label lu → Bool)
  (isRoscoe : Bool)
  (ref : DRefusal P isRoscoe X)
  → DRefusal P' isRoscoe X

bisimDRefusalS (terminate x) (terminate x1) PS PP' X isRoscoe ref tickIncl = ref tickIncl
bisimDRefusalS (terminate x) (node Q) PS (eqterminate (termeqnode terequivP)) X isRoscoe
  drefusal (λ e → ⊥-elim (noExtChoice terequivP e))(λ t → ...)
bisimDRefusalS {lu}{c} (node P) (terminate x) PS
  (eqterminater (termeqnode terequivP)) X isRoscoe (drefusal noextChInX noTerm) tickIncl
  = noTerm tickInc (lemBisimDRefusalAuxS c x P (hasTauOrTickNoTau terequivP))
bisimDRefusalS (node P) (node P') PS (eqnode bisimQQ') X isRoscoe ref =
  bisimDRefusal+S P P' PS bisim

```

mutual

```

lemmaxxx1S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)
  → (Q : Process ∞ {lu} c)
  → (divQ : DivergentProcess ∞ {lu} c Q)
  → BisimForNextP result (inj1 Q)
  → Process ∞ {lu} c

lemmaxxx1S (inj1 Q') Q divQ BisimResultQ = Q'
lemmaxxx1S (inj2 y) Q divQ BisimResultQ = ⊥-elim (lemmaDivNotTermequiv Q divQ y B)

lemmaxxx2S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
  → (divQ : DivergentProcess ∞ {lu} c Q)
  → (bisimForNext : BisimForNextP result (inj1 Q))
  → (trp : TrP {lu} l result P)
  → TrP {lu} l (inj1 (lemmaxxx1S result Q divQ bisimForNext)) P

lemmaxxx2S (inj1 x) l P Q divQ bisimForNext trp = trp
lemmaxxx2S (inj2 y) l P Q divQ bisimForNext trp = ⊥-elim (lemmaDivNotTermequiv Q divQ y B)

lemmaxxx2+S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)
  → (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)

```

```

→ (divQ : DivergentProcess ∞ {lu} c Q)
→ (bisimForNext : BisimForNextP result (inj1 Q))
→ (trp+ : TrP+ {lu} l result P)
→ TrP+ {lu} l (inj1 (lemmaxxx1S result Q divQ bisimForNext)) P
lemmaxxx2+S (inj1 x) l P Q divQ bisimForNext trp+ = trp+
lemmaxxx2+S (inj2 y) l P Q divQ bisimForNext trp+ = ⊥-elim (lemmaDivNotTermequiv Q divQ y bisimForNext)

lemmaxxx3S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (l : List (Label lu))(Q : Process ∞ {lu} c)
→ (divQ : DivergentProcess ∞ {lu} c Q)
→ (bisimForNext : BisimForNextP result (inj1 Q))
→ Bisimw (lemmaxxx1S result Q divQ bisimForNext) Q
lemmaxxx3S (inj1 Q') l Q divQ bisimForNext = bisimForNext
lemmaxxx3S (inj2 y) l Q divQ bisimForNext = ⊥-elim (lemmaDivNotTermequiv Q divQ y bisimForNext)

```

mutual

```

lemmayyy1S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true X
→ BisimForNextP result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1S (inj1 Q') Q stab X x x1 = Q'
lemmayyy1S (inj2 y) Q stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy1+S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊔ ChoiceSet c)
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ DRefusal {lu}{c} Q true X
→ BisimForNextP result (inj1 Q)
→ Process ∞ {lu} c
lemmayyy1+S (inj1 Q') Q stab X x x1 = Q'
lemmayyy1+S (inj2 y) Q stab X dref termequivQ =
  ⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy2S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet
→ (l : List (Label lu))(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ TrP {lu} l result P
→ TrP {lu} l (inj1 (lemmayyy1S result Q stab X dref bisim)) P
lemmayyy2S (inj1 Q') l P Q stab X dref bisim tr = tr
lemmayyy2S (inj2 y) l P Q stab X dref termequivQ x =
⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy2+S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet c)
→ (l : List (Label lu))(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
→ (stab+ : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ TrP+ {lu} l result P
→ TrP+ {lu} l (inj1 (lemmayyy1S result Q stab+ X dref bisim)) P
lemmayyy2+S (inj1 Q') l P Q stab X dref bisim tr = tr
lemmayyy2+S (inj2 y) l P Q stab X dref termequivQ x =
⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemmayyy3S : {lu : LUniv}{c : Choice} → (result : Process ∞ {lu} c ⊕ ChoiceSet
→ (Q : Process ∞ {lu} c)
→ (stab : stable Q)
→ (X : Label lu → Bool)
→ (dref : DRefusal {lu}{c} Q true X)
→ (bisim : BisimForNextP result (inj1 Q))
→ Bisimw (lemmayyy1S result Q stab X dref bisim) Q
lemmayyy3S (inj1 Q') Q stab X dref bisim = bisim
lemmayyy3S (inj2 y) Q stab X dref termequivQ =
⊥-elim (lemmaDrefusalStableNotTermequiv Q X dref stab y termequivQ)

```

```

lemEqListS : {A : Set}{l : List A} → l ++ [] ≡ l
lemEqListS [] = refl
lemEqListS (x :: l) = cong (λ l' → x :: l') (lemEqListS l)

stableNotTerminateEquivauxS : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (hasTauOrTickNoTau : ChoiceSet (I P) ⊔
    (¬ (ChoiceSet (I P)) × ChoiceSet (T P)))
  (stabSch : stableSch+ P)
  (notick : noTickIfRoscoe+ true P)
  → ⊥

stableNotTerminateEquivauxS P x (inj1 int) stabSch notick = stabSch int
stableNotTerminateEquivauxS P x (inj2 (noint ,, tick)) stabSch notick = notick tick

stableNotTerminateEquivS : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (x : ChoiceSet c)
  (terequiv : TerminateEquivalent x P)
  (stab : stable P)
  → ⊥

stableNotTerminateEquivS (terminate x) x1 terequiv stab = stab _
stableNotTerminateEquivS (node P) x (termeqnode terequivP) (stabSch ,, notick)
  = stableNotTerminateEquivauxS P x (hasTauOrTickNoTau terequivP) stabSch notick

noIntNoTerImpliesNoTermTraceS : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (x : ChoiceSet c)
  (tr : TrP+ [] (inj2 x) P)
  (noInt : ¬ (ChoiceSet (I P)))
  (noTer : ¬ (ChoiceSet (T P)))
  → ⊥

noIntNoTerImpliesNoTermTraceS P x (intc .[] .(inj2 x) int x2) noInt noTer = noInt int
noIntNoTerImpliesNoTermTraceS P .(PT P ter') (terc ter') noInt noTer = noTer ter'

mutual
bisimwStableToNoTickS : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (stabP' : stable P')
  (stabSchP : stableSch P)

```

```

→ noTickIfRoscoe true P
bisimwStableToNoTickS (terminate x) P' (eqterminate terequiv) stabP' stabSchP =
    stableNotTerminateEquivS P'
bisimwStableToNoTickS (terminate x) .(terminate _) (eqterminater terequiv) stabP' stabSchP
bisimwStableToNoTickS (node P) (terminate x) PP' stabP' stabSchP t = stabP' _
bisimwStableToNoTickS (node P) (node P') (eqnode PP') (noint „ noterP') noterP ter'
    = noIntNoTerImpliesNoTermTraceS P' (PT P ter') (bisimTtr PP' ter') noint noterP

```

mutual

--@BEGIN@bisimRefusalros

```

bisimRefusalrosS : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
    (P' : Process ∞ {lu} c)
    (PP' : Bisimw {∞} P P') (l : List (Label lu))
    (X : Label lu → Bool)
    (fail : failure P' l true X)
    → failure P l true X
bisimRefusalrosS {lu}{c} P P' PP' l X
    (stableFail (stableFp Q' tr' stab' drefuse'))
    = (stableFail (stableFp Qhat trhat₂
        (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat stabNoTick )
        drefusehat))

```

where

```

Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
Qcom = bisimTraceTrP₁ P P' PP' l (inj₁ Q') tr'

trcom : TrP {lu} l (bisimTraceTrP₁ P P' PP' l
    (inj₁ Q') tr') P
trcom = bisimTraceTrP₂ P P' PP' l (inj₁ Q') tr'

QQ'com : BisimForNextP (bisimTraceTrP₁ P P' PP' l
    (inj₁ Q') tr') (inj₁ Q')
QQ'com = bisimTraceTrP₃ P P' PP' l (inj₁ Q') tr'

Q : Process ∞ {lu} c
Q = lemmayyy₁S Qcom Q' stab' X drefuse' QQ'com

tr : TrP {lu} l (inj₁ Q) P
tr = lemmayyy₂S Qcom l P Q' stab' X drefuse' QQ'com trcom

```

```

QQ' : Bisimw Q Q'
QQ' = lemmayyy3S Qcom Q' stab' X drefuse' QQ'com

Qhat : Process ∞ {lu} c
Qhat = nonDivBecomeStable1S c Q
      (bisimStableImpliesNotDivergentS c Q Q' QQ' stab'
       (stableImpliesNonDivS Q' stab'))

trhat : TrP {lu} [] (inj1 Qhat) Q
trhat = nonDivBecomeStable2S c Q
      (bisimStableImpliesNotDivergentS c Q Q' QQ' stab'
       (stableImpliesNonDivS Q' stab'))

QhatQ' : Bisimw Qhat Q'
QhatQ' = bisimPPWithEmptyTrS Q Q' QQ' stab'
      (bisimStableImpliesNotDivergentS c Q Q' QQ' stab'
       (stableImpliesNonDivS Q' stab')) trhat

stabSchQhat : stableSch Qhat
stabSchQhat = nonDivBecomeStable3S c Q
      (bisimStableImpliesNotDivergentS c Q Q' QQ' stab'
       (stableImpliesNonDivS Q' stab'))

stabNoTick : noTickIfRoscoe true Qhat
stabNoTick = bisimwStableToNoTickS Qhat Q' QhatQ' stab' stabSchQhat

trhat1 : TrP {lu} (l ++ []) (inj1 Qhat) P
trhat1 = trPAppendTrw c P Q l [] (inj1 Qhat) tr trhat

eq1 : (l ++ []) ≡ l
eq1 = lemEqListS l

trhat2 : TrP {lu} l (inj1 Qhat) P
trhat2 = subst (λ l' → TrP {lu} l' (inj1 Qhat) P) eq1 trhat1

drefusehat : DRefusal Qhat true X
drefusehat = bisimDRefusalS Q' Qhat stab'
      (BismwSym Qhat Q' QhatQ') X true drefuse'

bisimRefusalrosS {lu}{c} P P' PP' l X

```

$$\begin{aligned} & (\text{divergentFailure } (\text{trdiv } Q' \text{ trp}' \text{ divq}')) \\ & = (\text{divergentFailure } (\text{trdiv } Q \text{ tr divp})) \end{aligned}$$

where

$$\begin{aligned} \text{Qcom} & : \text{Process } \infty \{lu\} \text{ } c \uplus \text{ChoiceSet } c \\ \text{Qcom} & = \text{bisimTraceTrP}_1 \text{ } P \text{ } P' \text{ } PP' \text{ } l \text{ } (\text{inj}_1 \text{ } Q') \text{ } \text{trp}' \\ \\ \text{trcom} & : \text{TrP } \{lu\} \text{ } l \text{ } (\text{bisimTraceTrP}_1 \text{ } P \text{ } P' \text{ } PP' \text{ } l \text{ } (\text{inj}_1 \text{ } Q') \text{ } \text{trp}') \text{ } P \\ \text{trcom} & = \text{bisimTraceTrP}_2 \text{ } P \text{ } P' \text{ } PP' \text{ } l \text{ } (\text{inj}_1 \text{ } Q') \text{ } \text{trp}' \\ \\ \text{QQ'com} & : \text{BisimForNextP} \\ & \quad (\text{bisimTraceTrP}_1 \text{ } P \text{ } P' \text{ } PP' \text{ } l \text{ } (\text{inj}_1 \text{ } Q') \text{ } \text{trp}') \text{ } (\text{inj}_1 \text{ } Q') \\ \text{QQ'com} & = \text{bisimTraceTrP}_3 \text{ } P \text{ } P' \text{ } PP' \text{ } l \text{ } (\text{inj}_1 \text{ } Q') \text{ } \text{trp}' \\ \\ Q & : \text{Process } \infty \{lu\} \text{ } c \\ Q & = \text{lemmaxxx}_1 \text{S } Qcom \text{ } Q' \text{ } \text{divq}' \text{ } QQ'com \\ \\ \text{tr} & : \text{TrP } \{lu\} \text{ } l \text{ } (\text{inj}_1 \text{ } Q) \text{ } P \\ \text{tr} & = \text{lemmaxxx}_2 \text{S } Qcom \text{ } l \text{ } P \text{ } Q' \text{ } \text{divq}' \text{ } QQ'com \text{ } \text{trcom} \\ \\ QQ' & : \text{Bisimw } Q \text{ } Q' \\ QQ' & = \text{lemmaxxx}_3 \text{S } Qcom \text{ } l \text{ } Q' \text{ } \text{divq}' \text{ } QQ'com \\ \\ Q'Q & : \text{Bisimw } Q' \text{ } Q \\ Q'Q & = \text{BismwSym } Q \text{ } Q' \text{ } QQ' \\ \\ \text{divp} & : \text{DivergentProcess } \infty \{lu\} \text{ } c \text{ } Q \\ \text{divp} & = \text{bisimImpliesDivergentPreservS } c \text{ } Q' \text{ } Q \text{ } Q'Q \text{ } \text{divq}' \end{aligned}$$

--@END

--@BEGIN@bisimRefusalrosplus

mutual

$$\begin{aligned} \text{bisimRefusalros+S} & : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} \text{ } c) \\ & \quad (P' : \text{Process+ } \infty \{lu\} \text{ } c) \\ & \quad (PP' : \text{Bisimw+ } \{\infty\} \text{ } P \text{ } P') \\ & \quad (l : \text{List } (\text{Label } lu)) \\ & \quad (X : \text{Label } lu \rightarrow \text{Bool}) \\ & \quad (\text{fail} : \text{failure+ } P' \text{ } l \text{ } \text{true} \quad X) \end{aligned}$$

$$\begin{aligned} & \rightarrow \text{failure}+ P \text{ l } \text{true} \quad X \\ \text{bisimRefusalros}+S \{lu\}\{c\} P P' PP' \text{ l } X \\ & (\text{stableFail} (\text{stableFp} Q' \text{ tr}' \text{ stab}' \text{ drefuse}')) \\ & = (\text{stableFail} (\text{stableFp} \text{Qhat} \text{ trhat}_2 \\ & (\text{stabSchNoTickIfRos2StablePar} \text{Qhat} \text{ true} \text{ stabSchQhat} \text{ stabNoTick}) \text{ drefusehat})) \end{aligned}$$

where

$$\begin{aligned} \text{Qcom} & : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet} c \\ \text{Qcom} & = \text{bisimTraceTrP}_1+ P P' PP' \text{ l } (\text{inj}_1 Q') \text{ tr}' \end{aligned}$$

$$\begin{aligned} \text{trcom} & : \text{TrP}+ \text{ l } (\text{bisimTraceTrP}_1+ P P' PP' \text{ l } (\text{inj}_1 Q') \text{ tr}') P \\ \text{trcom} & = \text{bisimTraceTrP}_2+ P P' PP' \text{ l } (\text{inj}_1 Q') \text{ tr}' \end{aligned}$$

$$\begin{aligned} \text{QQ'com} & : \text{BisimForNextP} \\ & (\text{bisimTraceTrP}_1+ P P' PP' \text{ l } (\text{inj}_1 Q') \text{ tr}') (\text{inj}_1 Q') \\ \text{QQ'com} & = \text{bisimTraceTrP}_3+ P P' PP' \text{ l } (\text{inj}_1 Q') \text{ tr}' \end{aligned}$$

$$\begin{aligned} Q & : \text{Process} \infty \{lu\} c \\ Q & = \text{lemmayyy}_1S \text{Qcom} Q' \quad \text{stab}' X \text{drefuse}' \text{QQ'com} \end{aligned}$$

$$\begin{aligned} \text{tr} & : \text{TrP}+ \{lu\} \text{ l } (\text{inj}_1 Q) P \\ \text{tr} & = \text{lemmayyy}_2+S \text{Qcom} \text{ l } P Q' \quad \text{stab}' X \text{drefuse}' \text{QQ'com} \text{trcom} \end{aligned}$$

$$\begin{aligned} \text{QQ'} & : \text{Bisimw} Q Q' \\ \text{QQ'} & = \text{lemmayyy}_3S \text{Qcom} Q' \quad \text{stab}' X \text{drefuse}' \text{QQ'com} \end{aligned}$$

$$\begin{aligned} \text{Qhat} & : \text{Process} \infty \{lu\} c \\ \text{Qhat} & = \text{nonDivBecomeStable}_1S c Q \\ & (\text{bisimStableImpliesNotDivergentS} c Q Q' \text{QQ}' \text{stab}' \\ & (\text{stableImpliesNonDivS} Q' \text{stab}')) \end{aligned}$$

$$\begin{aligned} \text{trhat} & : \text{TrP} \{lu\} [] (\text{inj}_1 \text{Qhat}) Q \\ \text{trhat} & = \text{nonDivBecomeStable}_2S c Q \\ & (\text{bisimStableImpliesNotDivergentS} c Q Q' \text{QQ}' \text{stab}' \\ & (\text{stableImpliesNonDivS} Q' \text{stab}')) \end{aligned}$$

$$\begin{aligned} \text{QhatQ'} & : \text{Bisimw} \text{Qhat} Q' \\ \text{QhatQ'} & = \text{bisimPPWithEmptyTrS} Q Q' \text{QQ}' \text{stab}' \\ & (\text{bisimStableImpliesNotDivergentS} c Q Q' \text{QQ}' \text{stab}' \\ & (\text{stableImpliesNonDivS} Q' \text{stab}')) \text{trhat} \end{aligned}$$

$$\text{stabSchQhat} : \text{stableSch} \text{Qhat}$$

$$\begin{aligned} \text{stabSchQhat} &= \text{nonDivBecomeStable}_3 S \ c \ Q \\ &\quad (\text{bisimStableImpliesNotDivergentS} \ c \ Q \ Q' \ QQ' \ \text{stab}' \\ &\quad \quad (\text{stableImpliesNonDivS} \ Q' \ \text{stab}')) \end{aligned}$$

$$\begin{aligned} \text{stabNoTick} &: \text{noTickIfRoscoe} \ \text{true} \ Qhat \\ \text{stabNoTick} &= \text{bisimwStableToNoTickS} \ Qhat \ Q' \ QhatQ' \ \text{stab}' \ \text{stabSchQhat} \end{aligned}$$

$$\begin{aligned} \text{trhat}_1 &: \text{TrP} + \{lu\} \ (l \ ++ \ []) \ (\text{inj}_1 \ Qhat) \ P \\ \text{trhat}_1 &= \text{trPAppendTrw} + \ c \ P \ Q \ l \ \quad [] \ (\text{inj}_1 \ Qhat) \ \text{tr} \ \text{trhat} \end{aligned}$$

$$\begin{aligned} \text{eqI} &: \quad (l \ ++ \ []) \equiv l \\ \text{eqI} &= \text{lemEqListS} \ l \end{aligned}$$

$$\begin{aligned} \text{trhat}_2 &: \text{TrP} + \{lu\} \ l \ (\text{inj}_1 \ Qhat) \ P \\ \text{trhat}_2 &= \text{subst} \ (\lambda \ l' \rightarrow \text{TrP} + \{lu\} \ l' \ (\text{inj}_1 \ Qhat) \ P) \ \text{eqI} \ \text{trhat}_1 \end{aligned}$$

$$\begin{aligned} \text{drefusehat} &: \quad \text{DRefusal} \ Qhat \ \text{true} \ X \\ \text{drefusehat} &= \quad \text{bisimDRefusalS} \ Q' \ Qhat \ \text{stab}' \\ &\quad (\text{BismwSym} \ Qhat \ Q' \ QhatQ') \ X \ \text{true} \ \text{drefuse}' \end{aligned}$$

$$\begin{aligned} \text{bisimRefusalros+S} \ \{lu\} \ \{c\} \ P \ P' \ PP' \ l \ X \\ &\quad (\text{divergentFailure} \ (\text{trdiv} \ Q' \ \text{trp}' \ \text{divq}')) \\ &= (\text{divergentFailure} \ (\text{trdiv} \ Q \ \text{tr} \ \text{divp})) \end{aligned}$$

where

$$\begin{aligned} \text{Qcom} &: \quad \text{Process} \ \infty \ \{lu\} \ c \ \uplus \ \text{ChoiceSet} \ c \\ \text{Qcom} &= \text{bisimTraceTrP}_1 + \{lu\} \ P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \end{aligned}$$

$$\begin{aligned} \text{trcom} &: \text{TrP} + \{lu\} \ l \ (\text{bisimTraceTrP}_1 + P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}') \ P \\ \text{trcom} &= \text{bisimTraceTrP}_2 + P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \end{aligned}$$

$$\begin{aligned} \text{QQ'com} &: \text{BisimForNextP} \\ &\quad (\text{bisimTraceTrP}_1 + P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}') \ (\text{inj}_1 \ Q') \\ \text{QQ'com} &= \text{bisimTraceTrP}_3 + P \ P' \ PP' \ l \ (\text{inj}_1 \ Q') \ \text{trp}' \end{aligned}$$

$$\begin{aligned} Q &: \text{Process} \ \infty \ \{lu\} \ c \\ Q &= \text{lemmaxx}_1 S \ Qcom \ Q' \ \text{divq}' \ \text{QQ'com} \end{aligned}$$

$$\text{tr} : \text{TrP} + \{lu\} \ l \ (\text{inj}_1 \ Q) \ P$$

```

tr = lemmaxx2+S Qcom l P Q' divq' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmaxx3S Qcom l Q' divq' QQ'com

Q'Q : Bisimw Q' Q
Q'Q = BismwSym Q Q' QQ'

divp : DivergentProcess ∞ {lu} c Q
divp = bisimImpliesDivergentPreservS c Q' Q Q'Q divq'

--@END

--@BEGIN@bisimImFdiTwo

bisimImFDI2S : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  → P ⊑fdi2ros P'
bisimImFDI2S {lu}{c} P P' PP' = bisimRefusalrosS P P' PP'

bisimImFDI2rS : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
  (P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  → P' ⊑fdi2ros P
bisimImFDI2rS {lu}{c} P P' PP' = bisimImFDI2S P' P (BismwSym P P' PP')

--@END

bisimImFDI2+S : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  → P ⊑fdi2ros+ P'
bisimImFDI2+S {lu}{c} P P' PP' = bisimRefusalros+S P P' PP'

bisimImFDI2r+S : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
  (P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  → P' ⊑fdi2ros+ P
bisimImFDI2r+S {lu}{c} P P' PP' = bisimImFDI2+S P' P (BismwSym+ P P' PP')

```

A.15 bisimImpliesBisim.agda

```

--@PREFIX@bisimImpliesBisim

module bisimImpliesBisim where

open import process
open import choiceSetU
open import labelUniv
open import Size
open import bisimilarity

mutual
--@BEGIN@BismwSymTheo

  BismwSym $\infty$  : {i : Size}{lu : LUniv}
    {c : Choice}
    (P P' : Process $\infty$   $\infty$  {lu} c)
    (PP' : Bisimw $\infty$  {i} P P')
     $\rightarrow$  Bisimw $\infty$  {i} P' P

  BismwSym : {i : Size}{lu : LUniv}{c : Choice}
    (P P' : Process  $\infty$  {lu} c)
    (PP' : Bisimw {i} P P')
     $\rightarrow$  Bisimw {i} P' P

  BismwSym+ : {i : Size}{lu : LUniv}{c : Choice}
    (P P' : Process+  $\infty$  {lu} c)
    (PP' : Bisimw+ {i} P P')
     $\rightarrow$  Bisimw+ {i} P' P

--@END

--@BEGIN@BismwSyminf

```

```

forceB (BismwSym $\infty$  P P' PP') = BismwSym (forceP P)
                                   (forceP P') (forceB PP')

--@END

--@BEGIN@BismwSym

BismwSym (terminate x) (terminate .x) (eqterminate termeqterm) =
    eqterminate termeqterm
BismwSym (terminate x) (terminate .x) (eqterminater termeqterm) =
    eqterminater termeqterm
BismwSym (terminate x) (node P') (eqterminate (termeqnode terequivP)) =
    eqterminater (termeqnode terequivP)
BismwSym (node P) (terminate x) (eqterminater (termeqnode terequivP)) =
    eqterminate (termeqnode terequivP)
BismwSym (node P) (node P') (eqnode PP') = eqnode (BismwSym+ P P' PP')

--@END

--@BEGIN@BismwSymplus

bisimdiv (BismwSym+ P P' PP') = bisimdivr PP'
nondiv+ (BismwSym+ P P' PP') = nondiv+r PP'
bisimEP' (BismwSym+ P P' PP') = bisimEP'r PP'
bisimEtr (BismwSym+ P P' PP') = bisimEtrr PP'
bisimEnext (BismwSym+ P P' PP') e =
    BismwSym $\infty$  (bisimEP'r PP' e) (PE P' e) (bisimEnextr PP' e)
bisimIP' (BismwSym+ P P' PP') = bisimIP'r PP'
bisimltr (BismwSym+ P P' PP') = bisimltrr PP'
bisimlnext (BismwSym+ P P' PP') e =
    BismwSym $\infty$  (bisimIP'r PP' e) (PI P' e) (bisimlnextr PP' e)
bisimTtr (BismwSym+ P P' PP') = bisimTtrr PP'
bisimdivr (BismwSym+ P P' PP') = bisimdiv PP'
nondiv+r (BismwSym+ P P' PP') = nondiv+ PP'
bisimEP'r (BismwSym+ P P' PP') = bisimEP' PP'
bisimEtrr (BismwSym+ P P' PP') = bisimEtr PP'
bisimEnextr (BismwSym+ P P' PP') e =
    BismwSym $\infty$  (PE P e) (bisimEP' PP' e) (bisimEnext PP' e)
bisimIP'r (BismwSym+ P P' PP') = bisimIP' PP'
bisimltrr (BismwSym+ P P' PP') = bisimltr PP'

```

```

bisimInextr (BismwSym+ P P' PP') e =
    BismwSym∞ (PI P e) (bisimIP' PP' e) (bisimInextr PP' e)
bisimTtrr (BismwSym+ P P' PP') = bisimTtr PP'

--@END

```

A.16 bisimImpliesFDI.agda

```
--@PREFIX@bisimImpliesFDImain
```

```
module bisimImpliesFDI where
```

```

open import process
open import choiceSetU
open import Size
open import Data.List.Base
open import Data.Maybe
open import Data.Sum
open import TraceWithNextProcess
open import bisimilarity
open import bisimImpliesBisim
open import labelUniv
open import fdi
open import fdiRefusal
open import bisimForNextProcess
open import auxData
open import bisimImpliesTraceEquiv
open import bisimilarityProofs
open import RefWithoutSize
open import TraceWithoutSize
open import bisimSImpliesBisimw
open import fdi
open import Data.Empty
open import Data.Bool.Base
open import Data.Unit

```

```
--@BEGIN@bisimImTrD
```

```
bisimImTrD : {lu : LUniv} {c : Choice} (P : Process ∞ {lu} c)
```

```

(P' : Process ∞ {lu} c)
(PP' : Bisimw {∞} P P') (l : List (Label lu))
(TrD : TraceDivergent ∞ c l P')
→ TraceDivergent ∞ c l P
bisimImTrD {lu}{c} P P' PP' l (trdiv Q' trp' divp') = trdiv Q tr divp
where
  Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
  Qcom = bisimTraceTrP1 P P' PP' l (inj1 Q') trp'

  trcom : TrP l (bisimTraceTrP1 P P' PP' l
                  (inj1 Q') trp') P
  trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') trp'

  QQ'com : BisimForNextP (bisimTraceTrP1 P P' PP' l
                          (inj1 Q') trp') (inj1 Q')
  QQ'com = bisimTraceTrP3 P P' PP' l (inj1 Q') trp'

  Q : Process ∞ {lu} c
  Q = lemaxxx1 Qcom Q' divp' QQ'com

  tr : TrP l (inj1 Q) P
  tr = lemaxxx2 Qcom l P Q' divp' QQ'com trcom

  QQ' : Bisimw Q Q'
  QQ' = lemaxxx3 Qcom l Q' divp' QQ'com

  Q'Q : Bisimw Q' Q
  Q'Q = BismwSym Q Q' QQ'

  divp : DivergentProcess ∞ c Q
  divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divp'

```

--@END

mutual

```

infTrNotTerEquiv+ : {i : Size}{lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
                  (l : Stream {∞} (Label lu))
                  (tr : infTr+ {i} l P)
                  (x : ChoiceSet c)
                  (terequivP : TerminateEquivalent+ x P)

```

$\rightarrow \perp$
 $\text{infTrNotTerEquiv+ } \{i\} P l (\text{extc } .l x x_1 x_2) x_3 \text{ terequiv} P = \perp\text{-elim } (\text{noExtChoice } \text{terequiv} P x)$
 $\text{infTrNotTerEquiv+ } \{i\} P l (\text{intc } .l x tr) x' \text{ terequiv} P =$
 $\text{infTrNotTerEquiv}\infty \{i\} (\text{PI } P x) l tr x' (\text{onlyIntChoice } \text{terequiv} P)$
 $\text{infTrNotTerEquiv}\infty : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c)$
 $(l : \text{Stream } \{\infty\} (\text{Label } lu))$
 $(tr : \text{infTr } \{i\} l (\text{forcep } P \{\infty\}))$
 $(x : \text{ChoiceSet } c)$
 $(\text{terequiv} P : \text{TerminateEquivalent}\infty x P)$
 $\rightarrow \perp$
 $\text{infTrNotTerEquiv}\infty \{i\} P l tr x \text{ terequiv} P = \text{infTrNotTerEquiv } \{i\} (\text{forcep } P) l tr x \text{ terequiv} P$
 $\text{infTrNotTerEquiv} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process } \infty \{lu\} c)$
 $(l : \text{Stream } \{\infty\} (\text{Label } lu))$
 $(tr : \text{infTr } \{i\} l P)$
 $(x : \text{ChoiceSet } c)$
 $(\text{terequiv} P : \text{TerminateEquivalent } x P)$
 $\rightarrow \perp$
 $\text{infTrNotTerEquiv } (\text{terminate } x) l () .x \text{ termeqterm}$
 $\text{infTrNotTerEquiv } \{i\} (\text{node } P) l (\text{tnode } tr) x (\text{termeqnode } \text{terequiv} P)$
 $= \text{infTrNotTerEquiv+ } \{i\} P l tr x \text{ terequiv} P$

mutual

$\text{bisimImTrD+} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c) (P' : \text{Process+ } \infty \{lu\} c)$
 $(PP' : \text{Bisimw+ } \{\infty\} P P')$
 $(l : \text{List } (\text{Label } lu)) (TrD : \text{TraceDivergent+ } \infty c l P')$
 $\rightarrow \text{TraceDivergent+ } \infty c l P$
 $\text{bisimImTrD+ } \{lu\} \{c\} P P' PP' l (\text{trdiv } Q' trp' divp') = \text{trdiv } Q tr divp$
 where
 $\text{Qcom} : \text{Process } \infty \{lu\} c \uplus \text{ChoiceSet } c$
 $\text{Qcom} = \text{bisimTraceTrP}_1 + P P' PP' l (\text{inj}_1 Q') trp'$
 $\text{trcom} : \text{TrP+ } l (\text{bisimTraceTrP}_1 + P P' PP' l (\text{inj}_1 Q') trp') P$
 $\text{trcom} = \text{bisimTraceTrP}_2 + P P' PP' l (\text{inj}_1 Q') trp'$
 $\text{QQ'com} : \text{BisimForNextP } (\text{bisimTraceTrP}_1 + P P' PP' l (\text{inj}_1 Q') trp') (\text{inj}_1 Q')$
 $\text{QQ'com} = \text{bisimTraceTrP}_3 + P P' PP' l (\text{inj}_1 Q') trp'$



```

Q : Process ∞ {lu} c
Q = lemaxxx1 Qcom Q' divp' QQ'com

tr : TrP+ l (inj1 Q) P
tr = lemaxxx2+ Qcom l P Q' divp' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemaxxx3 Qcom l Q' divp' QQ'com

Q'Q : Bisimw Q' Q
Q'Q = BismwSym Q Q' QQ'

divp : DivergentProcess ∞ c Q
divp = bisimImpliesDivergentPreserv c Q' Q Q'Q divp'

--@BEGIN@bisimImFDIOneplus

bisimImFDI1 : {lu : LUniv}{c : Choice} (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  → P ⊑fdi1+ P'
bisimImFDI1 {lu}{c} P P' PP' = bisimImTrD+ P P' PP'

bisimImFDI1r : {lu : LUniv}{c : Choice} (P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  → P' ⊑fdi1+ P
bisimImFDI1r {lu}{c} P P' PP' = bisimImFDI1 P' P (BismwSym+ P P' PP')

--@END

```

A.17 bisimImpliesFDIPartTwo.agda

```

--@PREFIX@bisimImpliesFDIPartTwo
module bisimImpliesFDIPartTwo where

open import process
open import choiceSetU
open import Size
open import Data.List.Base

```



```

open import Data.Maybe
open import Data.Sum
open import TraceWithNextProcess
open import bisimilarity
open import bisimImpliesBisim
open import labelUniv
open import fdi
open import fdiRefusal
open import bisimForNextProcess
open import auxData
open import bisimImpliesTraceEquiv
open import bisimilarityProofs
open import RefWithoutSize
open import TraceWithoutSize
open import bisimSImpliesBisimw
open import fdi
open import Data.Empty
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import bisimImpliesFDI
open import labelUniv
open import Relation.Binary.PropositionalEquality

mutual
--@BEGIN@addtauTraceToInfiniteTrace

addτTraceToInfiniteTrace : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))
  (tr₁ : TrP [] (inj₁ Q) P)
  (tr₂ : infTr {i} l Q)
  → infTr {i} l P

addτTraceToInfiniteTrace .(terminate x) .(terminate x) l (empty x) tr₂ = tr₂
addτTraceToInfiniteTrace .(node P) Q l (tnode {[]} {-} {P} tr₁) tr₂ =
  tnode (addτTraceToInfiniteTrace+ P Q l tr₁ tr₂)

addτTraceToInfiniteTrace+ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))

```

```

      (tr1 : TrP+ [] (inj1 Q) P)
      (tr2 : infTr {i} l Q)
      → infTr+ {i} l P
addτTraceToInfiniteTrace+ P .(node P) l empty (tnode tr) = tr
addτTraceToInfiniteTrace+ P Q l (intc .[] .(inj1 Q) x tr1) tr2 =
  intc l x (addτTraceToInfiniteTrace (forcep (PI P x))
    Q l tr1 tr2)

```

```

addτTraceToInfiniteTrace∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (l : Stream (Label lu))
  (tr1 : TrP∞ [] (inj1 Q) P)
  (tr2 : infTr {i} l Q)
  → infTr∞ {↑ i} l P
forcepP (addτTraceToInfiniteTrace∞ P Q l tr1 tr2) =
  addτTraceToInfiniteTrace (forcep P) Q l tr1 tr2

```

--@END

```

addτTraceToInfiniteTrace∞∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (Q : Process∞ ∞ {lu} c)
  (l : Stream (Label lu))
  (tr1 : TrP∞ [] (inj1 (forcep Q)) P)
  (tr2 : infTr∞ {i} l Q)
  → infTr∞ {i} l P
forcepP (addτTraceToInfiniteTrace∞∞ P Q l tr1 tr2) {j} =
  addτTraceToInfiniteTrace (forcep P) (forcep Q) l tr1 (forcepP tr2)

```

mutual

```

bisimInfTr      : {i : Size}
  {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  (PP' : Bisimw {∞} P P')
  (l : Stream (Label lu))
  (tr : infTr {i} l P')
  → infTr {i} l P

```

```

bisimInfTr    {i} {lu} {c} (terminate x) .(node P) (eqterminate terequiv)
              l (tnode {.l} {P} tr) =
  ⊥-elim (infTrNotTerEquiv (node P) l (tnode {i} tr) x terequiv)
bisimInfTr    {i} {lu} {c} (node P) .(node P') (eqnode PP') l (tnode {.l}
{P'} tr) =
  tnode (bisimInfTr+ P P' PP' l tr)

--@BEGIN@bisimInfTrPlus

bisimInfTr+   : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
              (PP' : Bisimw+ {∞} P P')
              (l : Stream (Label lu))
              (tr : infTr+ {i} l P')
              → infTr+ {i} l P
bisimInfTr+ {i} {lu} {c} P P' PP' l (extc .l e eq tr) =
  bisimInfTr+aux∞ P Q Q' l (Lab P' e) eq tr1 QQ' tr
where
  Q : Process∞ ∞ c
  Q = bisimEP'r PP' e

  Q' : Process∞ ∞ c
  Q' = PE P' e

  tr1 : TrP+ (Lab P' e :: []) (inj1 (forcep Q)) P
  tr1 = bisimEtrr PP' e

  eqlab      : T' (Lab P' e ==| head l)
  eqlab      = sym==| {lu} {head l} {Lab P' e} eq

  tr1' : TrP+ (head l :: []) (inj1 (forcep Q)) P
  tr1' = transfLu {lu} (λ lab1 → TrP+ (lab1 :: []) (inj1
(forcep Q)) P) eqlab tr1

  QQ' : Bisimw∞ Q Q'
  QQ' = bisimEnextr PP' e

--@END

bisimInfTr+ {i} {lu} {c} P P' PP' l (intc .l int tr) =
  bisimτInfTr+aux P (forcep Q) (forcep Q') tr1 (forceB QQ') l tr
where

```

```

Q : Process $\infty$   $\infty$  c
Q = bisimIP' r    PP' int

Q' : Process $\infty$   $\infty$  c
Q' = PI P' int

tr1 : TrP+ [] (inj1 (forcep Q)) P
tr1 = bisimItrr    PP' int

QQ' : Bisimw $\infty$  Q Q'
QQ' = bisimInextr PP' int

```

```

bisimInfTr+aux $\infty$ p  : {i : Size}{lu : LUniv}{c : Choice}(P : Process  $\infty$  {lu} c)
  (Q : Process $\infty$   $\infty$  c)
  (Q' : Process $\infty$   $\infty$  c)
  (l : Stream (Label lu))
  (la : Label lu)
  (eqlab : T' (head l ==| la))
  (tr1 : TrP (la :: []) (inj1 (forcep Q)) P)
  (QQ' : Bisimw $\infty$  Q Q')
  (tr2 : infTr $\infty$  {i} (tail l) Q')
  → infTr {i} l P
bisimInfTr+aux $\infty$ p .(node P) Q Q' l la eqlab (tnode {P = P} tr1) QQ' tr2
  = tnode (bisimInfTr+aux $\infty$  P Q Q' l la eqlab tr1 QQ' tr2)

```

```
--@BEGIN@bisimInfTrAuxInfty
```

```

bisimInfTr+aux $\infty$   : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+  $\infty$  {lu} c)
  (Q : Process $\infty$   $\infty$  c)
  (Q' : Process $\infty$   $\infty$  c)
  (l : Stream (Label lu))
  (la : Label lu)
  (eqlab : T' (head l ==| la))
  (tr1 : TrP+ (la :: []) (inj1 (forcep Q)) P)
  (QQ' : Bisimw $\infty$  Q Q')
  (tr2 : infTr $\infty$  {i} (tail l) Q')
  → infTr+ {i} l P

```

```

bisimInfTr+aux∞ P Q Q' l .(Lab P x) eqlab
  (extc .[] .(inj1 (forcep Q)) x tr1)
  QQ' tr2
= extc l x eqlab
  (addTraceToInfiniteTrace∞∞ (PE P x) Q (tail l) tr1
    (bisimInfTr∞ Q Q' QQ' (tail l) tr2))

bisimInfTr+aux∞ P Q Q' l la eqlab
  (intc .(la :: []) .(inj1 (forcep Q)) x tr1) QQ' tr2 =
  intc l x
  (bisimInfTr+aux∞p (forcep (PI P x)) Q Q' l la eqlab tr1
    QQ' tr2)

--@END

--@BEGIN@bisimTauInfTrplusAux

bisimτInfTr+aux : {i : Size}{lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (Q : Process ∞ c)
  (Q' : Process ∞ c)
  (tr1 : TrP+ [] (inj1 Q) P)
  (QQ' : Bisimw Q Q')
  (l : Stream (Label lu))
  (tr2 : infTr {i} l Q')
  → infTr+ {i} l P

bisimτInfTr+aux P .(node P) .(node P')
  empty (eqnode PP') l (tnode {.l} {P'} tr) = bisimInfTr+ P P' PP' l tr

bisimτInfTr+aux P Q Q' (intc .[] .(inj1 Q) x tr1) QQ' l tr2 =
  intc l x (addTraceToInfiniteTrace (forcep (PI P x)) Q l tr1
    (bisimInfTr Q Q' QQ' l tr2))

--@END

-- this is the place where when we have external choice reflected from
-- the trace from P' to P we go down by in the trace starting from P
-- down in size as indicated by the forcetP
bisimInfTr∞ : {i : Size}{lu : LUniv}{c : Choice}(P P' : Process∞ ∞ {lu} c)
  (PP' : Bisimw∞ {∞} P P')
  (l : Stream (Label lu))
  (tr : infTr∞ {i} l P')

```



$$\text{forcetP} \quad (\text{bisimInfTr}_\infty \ P \ P' \ PP' \ l \ tr) \ \{j\} = \\ \text{bisimInfTr} \ (\text{forceP} \ P) \ (\text{forceP} \ P') \ (\text{forceB} \ PP') \ l \ (\text{forcetP} \ tr)$$

$$\text{bisimInfTr}_\infty' : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process}_\infty \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisimw}_\infty \ \{\infty\} \ P \ P') \\ (l : \text{Stream} \ (\text{Label} \ lu)) \\ (tr : \text{infTr}_\infty' \ \{i\} \ l \ P') \\ \rightarrow \text{infTr}_\infty' \ \{i\} \ l \ P$$

$$\text{bisimInfTr}_\infty' \quad P \ P' \ PP' \ l \ tr = \text{bisimInfTr} \ (\text{forceP} \ P) \ (\text{forceP} \ P') \ (\text{forceB} \ PP') \ l \ tr$$

$$\text{bisimImFDI}_3 : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \\ \rightarrow P \sqsubseteq_{\text{fdi}_3} P'$$

$$\text{bisimImFDI}_3 = \text{bisimInfTr} \ \{\infty\}$$

$$\text{bisimImFDI}_{3+} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process}_+ \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisimw}_+ \ \{\infty\} \ P \ P') \\ \rightarrow P \sqsubseteq_{\text{fdi}_{3+}} P'$$

$$\text{bisimImFDI}_{3+} = \text{bisimInfTr}_+ \ \{\infty\}$$

$$\text{bisimImFDI}_{3\infty} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process}_\infty \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisimw}_\infty \ \{\infty\} \ P \ P') \\ \rightarrow P \sqsubseteq_{\text{fdi}_{3\infty}} P'$$

$$\text{bisimImFDI}_{3\infty} = \text{bisimInfTr}_\infty \ \{\infty\}$$

mutual

$$\text{bisimFDI} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ (PP' : \text{Bisimw} \ \{\infty\} \ P \ P') \rightarrow P \sqsubseteq_{\text{fdi}} P'$$

$$\text{bisimFDI} \ (\text{terminate} \ x) \ (\text{terminate} \ .x) \ (\text{eqterminate} \ \text{term} \ \text{eqterm}) \\ = ((\ (\lambda \ l \ m \ tr \rightarrow tr) \ \text{,,} \ (\lambda \ l \ x_1 \rightarrow x_1)) \ \text{,,} \\ \ (\lambda \ l \ X \ x_1 \rightarrow x_1)) \ \text{,,} \ (\lambda \ l \ x \rightarrow x)$$

$$\text{bisimFDI} \ (\text{terminate} \ x) \ (\text{terminate} \ .x) \ (\text{eqterminater} \ \text{term} \ \text{eqterm}) \\ = ((\ (\lambda \ l \ m \ tr \rightarrow tr) \ \text{,,} \ (\lambda \ l \ x_1 \rightarrow x_1)) \ \text{,,}$$


```

      (λ l X x1 → x1)) ,, (λ l x → x)
bisimFDI (terminate x) (node P') (eqterminate (termeqnode terequivP))
=      (( (bisimTraceEq (terminate x) (node P')
      (eqterminate (termeqnode terequivP))) ,,
      (bisimImTrD (terminate x) (node P')
      (eqterminate (termeqnode terequivP)))) ,,
      (bisimImFDI2 (terminate x) (node P')
      (eqterminate (termeqnode terequivP)))) ,,
      (λ l → λ {(tnode tr) →
      ⊥-elim (infTrNotTerEquiv+ P' l tr x terequivP)})
bisimFDI (node P) (terminate x) (eqterminater (termeqnode terequivP))
=      (( (bisimTraceEq (node P) (terminate x)
      (eqterminater (termeqnode terequivP))) ,,
      (bisimImTrD (node P) (terminate x)
      (eqterminater (termeqnode terequivP)))) ,,
      (bisimImFDI2 (node P) (terminate x)
      (eqterminater (termeqnode terequivP)))) ,,
      (λ l → λ {()})
bisimFDI (node P) (node P') (eqnode bisimQQ')
=      (((bisimTraceEq (node P) (node P') (eqnode bisimQQ')) ,,
      (bisimImTrD (node P) (node P') (eqnode bisimQQ')) ,,
      (bisimImFDI2 (node P) (node P') (eqnode bisimQQ')) ,,
      (bisimImFDI3 (node P) (node P') (eqnode bisimQQ'))

```

mutual

```

bisimFDIr : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
      (PP' : Bisimw {∞} P P') → P' ⊑fdi P
bisimFDIr (terminate x) (terminate .x) (eqterminate termeqterm) =
      (((λ l m tr → tr) ,, (λ l x1 → x1)) ,, (λ l X x1 → x1)) ,,
      (λ l → λ {()})
bisimFDIr (terminate x) (terminate .x) (eqterminater termeqterm) =
      (((λ l m tr → tr) ,, (λ l x1 → x1)) ,, (λ l X x1 → x1)) ,,
      (λ l → λ {()})
bisimFDIr (terminate x) (node P') (eqterminate terequiv) =
      (((bisimTraceEq (node P') (terminate x) (eqterminater terequiv)) ,,
      (bisimImTrD (node P') (terminate x) (eqterminater terequiv))) ,,
      (bisimImFDI2 (node P') (terminate x) (eqterminater terequiv))) ,,
      (λ l → λ {()})
bisimFDIr (node P) (terminate x) (eqterminater (termeqnode terequivP)) =
      (((bisimTraceEq
      (terminate x) (node P)

```



```

      (eqterminate (termeqnode terequivP))) ,,
    (bisimImTrD (terminate x) (node P)
      (eqterminate (termeqnode terequivP))) ,,
    (bisimImFDI2 (terminate x) (node P)
      (eqterminate (termeqnode terequivP))) ,,
    (λ l → λ {(tnode tr) →
      ⊥-elim (infTrNotTerEquiv+ P l tr x terequivP)} )
bisimFDIr (node P) (node P') (eqnode bisimQQ') =
  (((bisimTraceEq (node P') (node P) (BismwSym (node P) (node P')
    (eqnode bisimQQ')))) ,,
    (bisimImTrD (node P') (node P) (BismwSym (node P) (node P')
      (eqnode bisimQQ')))) ,,
    (bisimImFDI2 (node P') (node P) (BismwSym (node P) (node P')
      (eqnode bisimQQ')))) ,,
    (bisimImFDI3 (node P') (node P) (BismwSym (node P) (node P')
      (eqnode bisimQQ'))))

bisimFDIImpEq : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c) (PP' : Bisimw {∞} P' P)
  → P' ≡fdi P
bisimFDIImpEq P P' PP' = (bisimFDI P' P PP') ,, (bisimFDIr P' P PP')

--@BEGIN@bisimFdiRef

bisimImFdiRef : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)
  (PP' : Bisimw+ {∞} P P')
  → P ⊆fdi+ P'
bisimImFdiRef P P' PP' = ((bisimTraceEq+ P P' PP' ,,
  bisimImFDI1 P P' PP') ,,
  bisimImFDI2+ P P' PP') ,,
  bisimImFDI3+ P P' PP'

--@END

--@BEGIN@bisimFdiRefTheo

```



```

bisimImFdiEquiv :      {lu : LUniv}{c : Choice}
                      (P P' : Process+ ∞ {lu} c)
                      (PP' : Bisimw+ {∞} P P')
                      → P ≡fdi+ P'

--@END

--@BEGIN@bisimFdiRefTheoProof

bisimImFdiEquiv P P' PP' = bisimImFdiRef P P' PP' „
                           bisimImFdiRef P' P (BismwSym+ P P' PP')

--@END

--@BEGIN@bisimsFdiRef

bisimslmFdiRef : {lu : LUniv}{c : Choice} (P P' : Process+ ∞ {lu} c)
                (PP' : Bisims+ {∞} P P')
                → P ≡fdi+ P'

bisimslmFdiRef P P' PP' = bisimImFdiRef P P' (bisimsToBismw+ P P' PP')

bisimslmFdiEquiv :      {lu : LUniv}{c : Choice} (P P' : Process+ ∞ {lu} c)
                      (PP' : Bisims+ {∞} P P')
                      → P ≡fdi+ P'

bisimslmFdiEquiv P P' PP' = bisimImFdiEquiv P P'
                           (bisimsToBismw+ P P' PP')

--@END

```

A.18 bisimImpliesRefinementInfiniteTraces.agda

```

--@PREFIX@bisimImpliesRefinementInfiniteTraces

```

```

module bisimImpliesRefinementInfiniteTraces where

```

```

open import process

```

```
open import choiceSetU
open import Size
open import Data.List.Base
open import Data.Maybe
open import Data.Sum
open import TraceWithNextProcess
open import bisimilarity
open import bisimImpliesBisim
open import labelUniv
open import fdi
open import fdiRefusal
open import bisimForNextProcess
open import auxData
open import bisimImpliesTraceEquiv
open import bisimilarityProofs
open import RefWithoutSize
open import TraceWithoutSize
open import bisimSImpliesBisimw
open import bisimImpliesFDI
```

A.19 bisimImpliesTraceEquiv.agda

```
--@PREFIX@bisimImpliesTrace

module bisimImpliesTraceEquiv where

open import process
open import choiceSetU
open import Size
open import Data.List.Base
open import Data.Maybe
open import Data.Sum
open import TraceWithNextProcess
open import TraceWithoutSize
open import RefWithoutSize
open import bisimilarity
open import bisimImpliesBisim
open import traceImpliesTraceP
open import bisimSImpliesBisimw
```

```

open import labelUniv
open import traceEquivalence
open import Data.Product
open import bisimSImpliesBisimw

mutual

--@BEGIN@bisimTraceTheo

bisimTraceEq $\infty$  : {lu : LUniv}{c : Choice}
                  (P P' : Process $\infty$   $\infty$  {lu} c)
                  (PP' : Bisimw $\infty$  { $\infty$ } P P')
                  → P  $\sqsubseteq_{\infty}$  P'

bisimTraceEq : {lu : LUniv}{c : Choice}
              (P P' : Process  $\infty$  c)
              (PP' : Bisimw { $\infty$ } P P')
              → P  $\sqsubseteq$  P'

bisimTraceEq+ : {lu : LUniv}{c : Choice}
               (P P' : Process+  $\infty$  c)
               (PP' : Bisimw+ { $\infty$ } P P')
               → P  $\sqsubseteq_+$  P'

--@END

--@BEGIN@bisimTraceinf

bisimTraceEq $\infty$  P P' PP' l m tr = bisimTraceEq (forceP P)
                                         (forceP P') (forceB PP' { $\infty$ }) l m tr

--@END

--@BEGIN@bisimTrace

bisimTraceEq .(terminate x) .(terminate x) (eqterminate termeqterm)
             .[] .(just x) (ter x) = ter x
bisimTraceEq P .(terminate x) (eqterminater terequivP)
             .[] .(just x) (ter x) =
             termEquivalentImpliesTrace P terequivP
bisimTraceEq .(terminate x) .(terminate x) (eqterminate termeqterm)

```

```

    .[] .nothing (empty x) = empty x
bisimTraceEq P .(terminate x) (eqterminater terequivP)
    .[] .nothing (empty x) =
    termEquivalentImpliesTraceEmpty P terequivP
bisimTraceEq .(terminate a) .(node P) (eqterminate {a} terEquivP)
    .l .x (tnode {l} {x} {P} tr) =
    termEquivalentTracelsTerTrace+ P l terEquivP tr
bisimTraceEq .(node Q) .(node Q') (eqnode {Q} {Q'} QQ') .[] .nothing
    (tnode {[]} {[]} {Q} empty) = tnode empty
bisimTraceEq {lu}{c} .(node P) .(node P') (eqnode {P} {P'} PP')
    .(Lab P' x :: l) .mc (tnode (extc l mc x tr2')) = tnode tr
where
  Q' : Process $\infty$   $\infty$  {lu} c
  Q' = PE P' x

  Q : Process $\infty$   $\infty$  {lu} c
  Q = bisimEP'r PP' x

  QQ' : Bisimw $\infty$  Q Q'
  QQ' = bisimEnextr PP' x

  tr1 : P  $\rightarrow$ +*[ Lab P' x :: [] ] (forcep Q)
  tr1 = bisimEtrr PP' x

  tr2' : Tr $\infty$  {lu} {c} l mc Q'
  tr2' = tr2'

  tr2 : Tr $\infty$  {lu} {c} l mc Q
  tr2 = bisimTraceEq $\infty$  {lu} {c} Q Q' QQ' l mc tr2'

  tr : Tr+ {lu} {c} (Lab P' x :: l) mc P
  tr = traceAppendTrw+ c P (forcep Q) (Lab P' x :: [])
    l mc tr1 tr2

bisimTraceEq {lu} {c} .(node P) .(node P') (eqnode {P} {P'} PP')
    l mc (tnode (intc .l .mc x tr2')) = tnode tr
where
  Q' : Process $\infty$   $\infty$  {lu} c
  Q' = PI P' x

```

```

Q : Process $\infty$   $\infty$  {lu} c
Q = bisimIP'r PP' x

QQ' : Bisimw $\infty$  Q Q'
QQ' = bisimInextr PP' x

tr1 : P  $\rightarrow$ +*[ [] ] (forcep Q)
tr1 = bisimltrr PP' x

tr2 : Tr $\infty$  {lu} {c} l mc Q
tr2 = bisimTraceEq $\infty$  {lu}{c} Q Q' QQ' l mc tr2'

tr : Tr+ {lu}{c} ([[] ++ l) mc P
tr = traceAppendTrw+ c P (forcep Q) [[] l mc tr1 tr2

bisimTraceEq {lu}{c} .(node P) .(node P') (eqnode {P} {P'} PP')
  .[] .(just (PT P' x)) (tnode {.[[]]} {.(just (PT P' x))}
    {P'} (terc x)) =
    tnode (trPtoTr+ [[] (inj2 (PT P' x)) P tr1)

where
tr1 : TrP+ [[] (inj2 (PT P' x)) P
tr1 = bisimTtrr PP' x

--@END

--@BEGIN@bisimTracePlus

bisimTraceEq+ P P' PP' .[] .nothing empty = empty
bisimTraceEq+ {lu}{c} P P' PP' .(Lab P' x :: l1) m (extc l1 .m x tr2') = tr

where
Q' : Process $\infty$   $\infty$  {lu} c
Q' = PE P' x

Q : Process $\infty$   $\infty$  {lu} c
Q = bisimEP'r PP' x

QQ' : Bisimw $\infty$  Q Q'
QQ' = bisimEnextr PP' x

```

```

tr1  : P →+*[ Lab P' x :: [] ] (forcep Q)
tr1  = bisimEtrr PP' x

```

```

tr2'' : Tr∞ {lu}{c} l1 m Q'
tr2'' = tr2'

```

```

tr2  : Tr∞ {lu}{c} l1 m Q
tr2  = bisimTraceEq∞ {lu}{c} Q Q' QQ' l1 m tr2''

```

```

tr : Tr+ {lu}{c} (Lab P' x :: l1) m P
tr = traceAppendTrw+ c P (forcep Q)
    (Lab P' x :: []) l1 m tr1 tr2

```

```

bisimTraceEq+ {lu}{c} P P' PP' l m (intc .l .m x tr2') = tr
where

```

```

Q' : Process∞ ∞ {lu} c
Q' = PI P' x

```

```

Q : Process∞ ∞ {lu} c
Q = bisimIP'r PP' x

```

```

QQ' : Bisimw∞ Q Q'
QQ' = bisimInextr PP' x

```

```

tr1  : P →+*[ [] ] (forcep Q)
tr1  = bisimltrr PP' x

```

```

tr2  : Tr∞ {lu}{c} l m Q
tr2  = bisimTraceEq∞ Q Q' QQ' l m tr2'

```

```

tr : Tr+ {lu}{c} ([+] ++ l) m P
tr = traceAppendTrw+ c P (forcep Q) [] l m tr1 tr2

```

```

bisimTraceEq+ P P' PP' .[] .(just (PT P' t)) (terc t) =
    trPtoTr+ [] (inj2 (PT P' t)) P tr1

```

```

where

```

```

tr1  : TrP+ [] (inj2 (PT P' t)) P
tr1  = bisimTtrr PP' t

```

```

--@END

```

mutual

--@BEGIN@SbisimTracinfnf

SbisimTraceEq ∞ : $\{lu : LUniv\}\{c : Choice\}(P P' : Process\infty \infty \{lu\} c)$
 $(PP' : Bisims\infty \{\infty\} P P') \rightarrow P \sqsubseteq\infty P'$

SbisimTraceEq ∞ $\{lu\}\{c\} P P' PP' l m tr = SbisimTraceEq$
 $(forcep P) (forcep P') (forceB PP') l m tr$

--@END

--@BEGIN@SbisimTrac

SbisimTraceEq : $\{lu : LUniv\}\{c : Choice\}(P P' : Process \infty c)$
 $(PP' : Bisims \{\infty\} P P') \rightarrow P \sqsubseteq P'$

SbisimTraceEq $.(terminate x) .(terminate x) eqterminate$
 $.[.] .(just x) (ter x) = ter x$

SbisimTraceEq $.(terminate x) .(terminate x) eqterminate$
 $.[.] .nothing (empty x) = empty x$

SbisimTraceEq $.(node P) .(node P') (eqnode \{P\} \{P'\} PP')$
 $.[.] .nothing (tnode empty) = tnode empty$

SbisimTraceEq $\{lu\}\{c\} .(node P) .(node P') (eqnode \{P\} \{P'\} PP')$
 $.(Lab P' x :: l) mc (tnode (extc l .mc x tr_2')) = tnode tr$

where

$Q' : Process\infty \infty \{lu\} c$

$Q' = PE P' x$

$Q : Process\infty \infty \{lu\} c$

$Q = bisimEP'r (bisimsToBismw+ P P' PP') x$

$QQ' : Bisimw\infty Q Q'$

$QQ' = bisimEnextr (bisimsToBismw+ P P' PP') x$

$tr_1 : P \rightarrow +*[Lab P' x :: []] (forcep Q)$

$tr_1 = bisimEtrr (bisimsToBismw+ P P' PP') x$

```
tr₂'' : Tr∞ {lu}{c} l mc Q'
tr₂'' = tr₂'
```

```
tr₂ : Tr∞ {lu}{c} l mc Q
tr₂ = bisimTraceEq∞ {lu}{c} Q Q' QQ' l mc tr₂''
```

```
tr : Tr+ {lu}{c} (Lab P' x :: l) mc P
tr = traceAppendTrw+ c P (forcep Q) (Lab P' x :: []) l mc tr₁ tr₂
```

```
SbisimTraceEq {lu}{c} .(node P) .(node P') (eqnode {P} {P'} PP')
  l mc (tnode (intc .l .mc x tr₂')) = tnode tr
```

where

```
Q' : Process∞ ∞ {lu} c
Q' = PI P' x
```

```
Q : Process∞ ∞ {lu} c
Q = bisimIP'r (bisimsToBismw+ P P' PP') x
```

```
QQ' : Bisimw∞ Q Q'
QQ' = bisimInextr (bisimsToBismw+ P P' PP') x
```

```
tr₁ : P →+*[ [] ] (forcep Q)
tr₁ = bisimLtrr (bisimsToBismw+ P P' PP') x
```

```
tr₂'' : Tr∞ {lu}{c} l mc Q'
tr₂'' = tr₂'
```

```
tr₂ : Tr∞ {lu}{c} l mc Q
tr₂ = bisimTraceEq∞ {lu}{c} Q Q' QQ' l mc tr₂''
```

```
tr : Tr+ {lu}{c} ([ [] ++ l) mc P
tr = traceAppendTrw+ c P (forcep Q) [ [] l mc tr₁ tr₂
```

```
SbisimTraceEq .(node P) .(node P') (eqnode {P} {P'} PP')
  .[ [] ] .(just (PT P' x)) (tnode (terc x)) =
  tnode (trPtoTr+ [ [] ] (inj₂ (PT P' x)) P tr₁)
```

where

```
tr₁ : TrP+ [ [] ] (inj₂ (PT P' x)) P
tr₁ = bisimTtrr (bisimsToBismw+ P P' PP') x
```

--@END

`{- We proof the reverse directions for the previous statements -}`

`--@BEGIN@bisimTraceReverse`

`bisimTraceEq ∞ r : {lu : LUniv}{c : Choice}(P P' : Process ∞ ∞ {lu} c)`
 `(PP' : Bisimw ∞ { ∞ } P P') \rightarrow P' \sqsubseteq_{∞} P`
`bisimTraceEq ∞ r P P' PP' = bisimTraceEq ∞ P' P (BismwSym ∞ P P' PP')`

`bisimTraceEq+r : {lu : LUniv}{c : Choice}(P P' : Process+ ∞ {lu} c)`
 `(PP' : Bisimw+ { ∞ } P P') \rightarrow P' \sqsubseteq_+ P`
`bisimTraceEq+r P P' PP' = bisimTraceEq+ P' P (BismwSym+ P P' PP')`

`bisimTraceEq r : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)`
 `(PP' : Bisimw { ∞ } P P') \rightarrow P' \sqsubseteq P`
`bisimTraceEq r P P' PP' = bisimTraceEq P' P (BismwSym P P' PP')`

`--@END`

`{- We prove that weak bisimilarity implies trace equivalence -}`

`--@BEGIN@bisimTraceEqTheo`

`bisimTraceEq ∞ = : {lu : LUniv}{c : Choice}`
 `(P P' : Process ∞ ∞ {lu} c)`
 `(PP' : Bisimw ∞ { ∞ } P P')`
 `\rightarrow P \equiv_{∞} P'`

`bisimTraceEq+= : {lu : LUniv}{c : Choice}`
 `(P P' : Process+ ∞ {lu} c)`
 `(PP' : Bisimw+ { ∞ } P P')`
 `\rightarrow P \equiv_+ P'`

`bisimTraceEq= : {lu : LUniv}{c : Choice}`
 `(P P' : Process ∞ {lu} c)`
 `(PP' : Bisimw { ∞ } P P')`
 `\rightarrow P \equiv P'`

`--@END`

```
--@BEGIN@bisimTraceEq
```

```
bisimTraceEq $\infty$ = P P' PP' = bisimTraceEq $\infty$  P P' PP' ,
                               bisimTraceEq $\infty$ r P P' PP'
```

```
bisimTraceEq+= P P' PP' = bisimTraceEq+ P P' PP' ,
                           bisimTraceEq+r P P' PP'
```

```
bisimTraceEq= P P' PP' = bisimTraceEq P P' PP' ,
                           bisimTraceEq r P P' PP'
```

```
--@END
```

```
{- We prove that strong bisimilarity implies trace equivalence -}
```

```
--@BEGIN@bisimTraceEqsTheo
```

```
bisimTraceEqs $\infty$ = : {lu : LUniv}{c : Choice}
                     (P P' : Process $\infty$   $\infty$  {lu} c)
                     (PP' : Bisims $\infty$  { $\infty$ } P P')
                      $\rightarrow$  P  $\equiv_{\infty}$  P'
```

```
bisimTraceEqs+= : {lu : LUniv}{c : Choice}
                  (P P' : Process+  $\infty$  {lu} c)
                  (PP' : Bisims+ { $\infty$ } P P')
                   $\rightarrow$  P  $\equiv_+$  P'
```

```
bisimTraceEqs= : {lu : LUniv}{c : Choice}
                 (P P' : Process  $\infty$  {lu} c)
                 (PP' : Bisims { $\infty$ } P P')
                  $\rightarrow$  P  $\equiv$  P'
```

```
--@END
```

```
--@BEGIN@bisimTraceEqs
```

```

bisimTraceEqs $\infty$ = P P' PP' =
    bisimTraceEq $\infty$ = P P' (bisimsToBismw $\infty$  P P' PP')

bisimTraceEqs+= P P' PP' =
    bisimTraceEq+= P P' (bisimsToBismw+ P P' PP')

bisimTraceEqs= P P' PP' =
    bisimTraceEq= P P' (bisimsToBismw P P' PP')

--@END

```

A.20 bisimLemFmap.agda

```

module bisimLemFmap where

```

```

open import process
open import choiceSetU
open import label
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import renamingResult
open import dataAuxFunction
open import fdi
open import bisimilarity
open import bisimSym
open import addTick
open import Data.Fin
open import labelUniv

mutual
  lemBisimFmap+ : {lu : LUniv}{c0 c1 c2 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
    → (g : ChoiceSet c1 → ChoiceSet c2)
    → (P : Process+  $\infty$  {lu} c0)

```

```

    → Bisims+ (fmap+ (g ∘ f) P) (fmap+ g (fmap+ f P))
bisim2E    (lemBisimFmap+ f g P) e = e
bisimELab  (lemBisimFmap+ f g P) e = refl
bisimENext (lemBisimFmap+ f g P) e = lemBisimFmap∞ f g (PE P e)
bisim2I    (lemBisimFmap+ f g P) e = e
bisimINext (lemBisimFmap+ f g P) e = lemBisimFmap∞ f g (PI P e)
bisim2T    (lemBisimFmap+ f g P) e = e
bisim2TEq  (lemBisimFmap+ f g P) e = refl
bisim2Er   (lemBisimFmap+ f g P) e = e
bisimELabr (lemBisimFmap+ f g P) e = refl
bisimENextr (lemBisimFmap+ f g P) e = lemBisimFmap∞ f g (PE P e)
bisim2Irr  (lemBisimFmap+ f g P) e = e
bisimINextr (lemBisimFmap+ f g P) e = lemBisimFmap∞ f g (PI P e)
bisim2Tr   (lemBisimFmap+ f g P) e = e
bisim2TEqr (lemBisimFmap+ f g P) e = refl

lemBisimFmap∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → (g : ChoiceSet c₁ → ChoiceSet c₂)
  → (P : Process∞ ∞ {lu} c₀)
  → Bisims∞ (fmap∞ (g ∘ f) P) (fmap∞ g (fmap∞ f P))
forceB (lemBisimFmap∞ f g P) = lemBisimFmap f g (forceP P)

lemBisimFmap : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → (g : ChoiceSet c₁ → ChoiceSet c₂)
  → (P : Process ∞ {lu} c₀)
  → Bisims (fmap (g ∘ f) P) (fmap g (fmap f P))
lemBisimFmap f g (terminate x) = eqterminate
lemBisimFmap f g (node x) = eqnode (lemBisimFmap+ f g x)

```

mutual

```

lemBisimFmap+R : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → (g : ChoiceSet c₁ → ChoiceSet c₂)
  → (P : Process+ ∞ {lu} c₀)
  → Bisims+ (fmap+ g (fmap+ f P)) (fmap+ (g ∘ f) P)
bisim2E    (lemBisimFmap+R f g P) e = e
bisimELab  (lemBisimFmap+R f g P) e = refl

```

```

bisimENext (lemBisimFmap+R f g P) e = lemBisimFmap∞R f g (PE P e)
bisim2I    (lemBisimFmap+R f g P) e = e
bisimINext (lemBisimFmap+R f g P) e = lemBisimFmap∞R f g (PI P e)
bisim2T    (lemBisimFmap+R f g P) e = e
bisim2TEq  (lemBisimFmap+R f g P) e = refl
bisim2Er   (lemBisimFmap+R f g P) e = e
bisimELabr (lemBisimFmap+R f g P) e = refl
bisimENextr (lemBisimFmap+R f g P) e = lemBisimFmap∞R f g (PE P e)
bisim2Itr  (lemBisimFmap+R f g P) e = e
bisimINextr (lemBisimFmap+R f g P) e = lemBisimFmap∞R f g (PI P e)
bisim2Tr   (lemBisimFmap+R f g P) e = e
bisim2TEqr (lemBisimFmap+R f g P) e = refl

```

```

lemBisimFmap∞R : {lu : LUniv} {c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → (g : ChoiceSet c₁ → ChoiceSet c₂)
  → (P : Process∞ ∞ {lu} c₀)
  → Bisims∞ (fmap∞ g (fmap∞ f P)) (fmap∞ (g ∘ f) P)
forceB (lemBisimFmap∞R f g P) = lemBisimFmapR f g (forceP P)

```

```

lemBisimFmapR : {lu : LUniv} {c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → (g : ChoiceSet c₁ → ChoiceSet c₂)
  → (P : Process ∞ {lu} c₀)
  → Bisims (fmap g (fmap f P)) (fmap (g ∘ f) P)
lemBisimFmapR f g (terminate x) = eqterminate
lemBisimFmapR f g (node x) = eqnode (lemBisimFmap+R f g x)

```

mutual

```

addTimeFmapBisimLemma+ : {lu : LUniv} {c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process+ ∞ {lu} c₀)
  (a : ChoiceSet c₁)

```

```

    → Bisims+ (addTimed✓+ (g a) (fmap+ (g ∘ f) P)) (fmap+ g (addTimed✓+ a (fmap+ f P) a))
bisim2E (addTimeFmapBisimLemma+ f g P a) e = e
bisimELab (addTimeFmapBisimLemma+ f g P a) e = refl
bisimENext (addTimeFmapBisimLemma+ f g P a) e = lemBisimFmap∞ f g (PE P e)
bisim2I (addTimeFmapBisimLemma+ f g P a) e = e
bisimINext (addTimeFmapBisimLemma+ f g P a) e = addTimeFmapBisimLemma∞ f g (PI P e) a
bisim2T (addTimeFmapBisimLemma+ f g P a) (inj1 x) = inj1 x
bisim2T (addTimeFmapBisimLemma+ f g P a) (inj2 y) = inj2 y
bisim2TEq (addTimeFmapBisimLemma+ f g P a) (inj1 x) = refl
bisim2TEq (addTimeFmapBisimLemma+ f g P a) (inj2 y) = refl
bisim2Er (addTimeFmapBisimLemma+ f g P a) e = e
bisimELabr (addTimeFmapBisimLemma+ f g P a) e = refl
bisimENextr (addTimeFmapBisimLemma+ f g P a) e = lemBisimFmap∞ f g (PE P e)
bisim2Itr (addTimeFmapBisimLemma+ f g P a) e = e
bisimINextr (addTimeFmapBisimLemma+ f g P a) e = addTimeFmapBisimLemma∞ f g (PI P e) a
bisim2Tr (addTimeFmapBisimLemma+ f g P a) (inj1 x) = inj1 x
bisim2Tr (addTimeFmapBisimLemma+ f g P a) (inj2 y) = inj2 y
bisim2TEqr (addTimeFmapBisimLemma+ f g P a) (inj1 x) = refl
bisim2TEqr (addTimeFmapBisimLemma+ f g P a) (inj2 y) = refl

```

```

addTimeFmapBisimLemma∞ : {lu : LUniv} {c0 c1 c2 : Choice}
  (f : ChoiceSet c0 → ChoiceSet c1)
  (g : ChoiceSet c1 → ChoiceSet c2)
  (P : Process∞ ∞ {lu} c0)
  (a : ChoiceSet c1)
  → Bisims∞ (addTimed✓∞ (g a) (fmap∞ (g ∘ f) P)) (fmap∞ g (addTimed✓∞ a (fmap f P) a))
forceB (addTimeFmapBisimLemma∞ {lu} f g P a) = addTimeFmapBisimLemma {lu} f g (forceP P)

```

```

addTimeFmapBisimLemma : {lu : LUniv} {c0 c1 c2 : Choice}
  (f : ChoiceSet c0 → ChoiceSet c1)
  (g : ChoiceSet c1 → ChoiceSet c2)
  (P : Process ∞ {lu} c0)
  (a : ChoiceSet c1)
  → Bisims (addTimed✓ (g a) (fmap (g ∘ f) P)) (fmap g (addTimed✓ a (fmap f P) a))
addTimeFmapBisimLemma f g (terminate x) a = eqnode (lem f g x a)
addTimeFmapBisimLemma f g (node x) a = eqnode (addTimeFmapBisimLemma+ f g x a)

```

```

lem : {lu : LUniv} {c0 c1 c2 : Choice}
  → (f : ChoiceSet c0 → ChoiceSet c1)
  → (g : ChoiceSet c1 → ChoiceSet c2)
  → (x : ChoiceSet c0)
  → (a : ChoiceSet c1)
  → Bisims+ {lu = lu} (fmap+ unifyA⊕A (2-✓+ (g a) ((g ∘ f) x)))
    (fmap+ g (fmap+ unifyA⊕A (2-✓+ a (f x))))
bisim2E (lem f g x a) ()
bisimELab (lem f g x a) ()
bisimENext (lem f g x a) ()
bisim2l (lem f g x a) ()
bisimlNext (lem f g x a) ()
bisim2T (lem f g x a) e = e
bisim2TEq (lem f g x a) zero = refl
bisim2TEq (lem f g x a) (suc zero) = refl
bisim2TEq (lem f g x a) (suc (suc ()))
bisim2Er (lem f g x a) ()
bisimELabr (lem f g x a) ()
bisimENextr (lem f g x a) ()
bisim2lr (lem f g x a) ()
bisimlNextr (lem f g x a) ()
bisim2Tr (lem f g x a) e = e
bisim2TEqr (lem f g x a) zero = refl
bisim2TEqr (lem f g x a) (suc zero) = refl
bisim2TEqr (lem f g x a) (suc (suc ()))

```

A.21 bisimSImpliesBisimw.agda

```
--@PREFIX@bisimSImpliesBisimw
```

```
module bisimSImpliesBisimw where
```

```

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Data.List

```

```

open import Data.Sum
open import TraceWithNextProcess
open import bisimilarity
open import bisimilarityProofs

mutual

--@BEGIN@bisimsToBismwTheo

bisimsToBismw $\infty$  : {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process $\infty$   $\infty$  {lu} c)
  → Bisims $\infty$  {i} P P'
  → Bisimw $\infty$  {i} P P'

bisimsToBismw : {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process  $\infty$  {lu} c)
  → Bisims {i} P P'
  → Bisimw {i} P P'

bisimsToBismw+ : {i : Size}{lu : LUniv}{c : Choice}
  (P P' : Process+  $\infty$  {lu} c)
  → Bisims+ {i} P P'
  → Bisimw+ {i} P P'

--@END

--@BEGIN@bisimsToBismwinf

forceB (bisimsToBismw $\infty$  {i} P P' PP') {j} = bisimsToBismw
  (forceP P) (forceP P') (forceB PP' {j})

--@END

--@BEGIN@bisimsToBismw

bisimsToBismw .(terminate a) .(terminate a) (eqterminate {a}) =
  eqterminate termeqterm
bisimsToBismw .(node Q) .(node Q') (eqnode {Q} {Q'} x) =
  eqnode (bisimsToBismw+ Q Q' x)

--@END

```

--@BEGIN@bisimsToBismwplus

```

bisimdiv    (bisimsToBismw+ P P' PP') = divLemBisims+ P P' PP'
nondiv+     (bisimsToBismw+ P P' PP') = nondivLemBisims+ P P' PP'
bisimEP'    (bisimsToBismw+ P P' PP') e = PE P' (bisim2E PP' e)
bisimEtr    (bisimsToBismw+ P P' PP') e rewrite (bisimELab PP' e) =
  extc [] (inj1 (forcep (PE P' (bisim2E PP' e))))
  (bisim2E PP' e)(reflTrP∞ (PE P' (bisim2E PP' e)))
bisimEnext  (bisimsToBismw+ P P' PP') e =
  bisimsToBismw∞ (PE P e) (PE P' (bisim2E PP' e))
  (bisimENext PP' e)
bisimIP'    (bisimsToBismw+ P P' PP') e = PI P' (bisim2I PP' e)
bisimltr    (bisimsToBismw+ P P' PP') e =
  intc [] ((inj1 (forcep (PI P' (bisim2I PP' e)))))
  (bisim2I PP' e)(reflTrP∞ ((PI P' (bisim2I PP' e))))
bisimInext  (bisimsToBismw+ P P' PP') e =
  bisimsToBismw∞ (PI P e) (PI P' (bisim2I PP' e))
  (bisimInext PP' e)
bisimTtr    (bisimsToBismw+ P P' PP') t rewrite (bisim2TEq PP' t) =
  terc (bisim2T PP' t)
bisimdivr   (bisimsToBismw+ P P' PP') = divLemBisims+r P P' PP'
nondiv+r    (bisimsToBismw+ P P' PP') = nondivLemBisims+r P P' PP'
bisimEP'r   (bisimsToBismw+ P P' PP') e = PE P (bisim2Er PP' e)
bisimEtrr   (bisimsToBismw+ P P' PP') e rewrite (bisimELabr PP' e) =
  extc [] (inj1 (forcep (PE P (bisim2Er PP' e)))))
  (bisim2Er PP' e)(reflTrP∞ (PE P (bisim2Er PP' e)))
bisimEnextr (bisimsToBismw+ P P' PP') e =
  bisimsToBismw∞ (PE P (bisim2Er PP' e)) (PE P' e)
  (bisimENextr PP' e)
bisimIP'r   (bisimsToBismw+ P P' PP') e = PI P (bisim2Ir PP' e)
bisimltrr   (bisimsToBismw+ P P' PP') e =
  intc [] (inj1 (forcep (PI P (bisim2Ir PP' e)))))
  (bisim2Ir PP' e)(reflTrP∞ (PI P (bisim2Ir PP' e)))
bisimInextr (bisimsToBismw+ P P' PP') e =
  bisimsToBismw∞ (PI P (bisim2Ir PP' e)) (PI P' e)
  ((bisimInextr PP' e))
bisimTtrr   (bisimsToBismw+ P P' PP') t rewrite (bisim2TEqr PP' t) =
  terc (bisim2Tr PP' t)

```

```
--@END
```

A.22 bisimSym.agda

```
--@PREFIX@bisimSym
```

```
module bisimSym where
```

```
open import process
open import choiceSetU
open import Size
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import bisimilarity
open import Agda.Builtin.Equality
open import labelUniv
```

```
mutual
```

```
--@BEGIN@BismwRefinfTheo
```

```

BismwRef $\infty$  : {i : Size}{lu : LUniv}{c : Choice}
              (P : Process $\infty$   $\infty$  {lu} c)
              → Bisimw $\infty$  {i} P P
BismwRef      : {i : Size}{lu : LUniv}{c : Choice}
              (P : Process  $\infty$  {lu} c)
              → Bisimw {i} P P
BismwRef+     : {i : Size}{lu : LUniv}{c : Choice}
              (P : Process+  $\infty$  {lu} c)
              → Bisimw+ {i} P P
```

```
--@END
```

```
--@BEGIN@BismwRefinf
```

```
forceB (BismwRef $\infty$  {i} {lu} P) {j} = BismwRef {j} {lu} (forceP P)
```

```
--@END
```

```
--@BEGIN@BismwRef
```

```
BismwRef (terminate x) = eqterminate term eqterm
BismwRef (node x) = eqnode (BismwRef+ x)
```

```
--@END
```

```
--@BEGIN@BismwRefplus
```

```
bisimdiv (BismwRef+ P) e = e
nondiv+ (BismwRef+ P) e = e
bisimEP' (BismwRef+ P) e = PE P e
bisimEtr (BismwRef+ P) e =
  extc [] (inj1 (forcep (PE P e))) e (reflTrP∞ (PE P e))
bisimEnext (BismwRef+ P) e = BismwRef∞ (PE P e)
bisimIP' (BismwRef+ P) i = PI P i
bisimltr (BismwRef+ P) e =
  intc [] (inj1 (forcep (PI P e))) e (reflTrP∞ (PI P e))
bisimInext (BismwRef+ P) e = BismwRef∞ (PI P e)
bisimTtr (BismwRef+ P) e = terc e
bisimdivr (BismwRef+ P) e = e
nondiv+r (BismwRef+ P) e = e
bisimEP'r (BismwRef+ P) e = PE P e
bisimEtrr (BismwRef+ P) e =
  extc [] (inj1 (forcep (PE P e))) e (reflTrP∞ (PE P e))
bisimEnexttr (BismwRef+ P) e = BismwRef∞ (PE P e)
bisimIP'r (BismwRef+ P) i = PI P i
bisimltrr (BismwRef+ P) e =
  intc [] (inj1 (forcep (PI P e))) e (reflTrP∞ (PI P e))
bisimInextr (BismwRef+ P) e = BismwRef∞ (PI P e)
bisimTtrr (BismwRef+ P) e = terc e
```

```
--@END
```

```
mutual
```

```
--@BEGIN@BismSRefTheo
```

```

BismsRef $\infty$  : {i : Size}{lu : LUniv}{c : Choice}
              (P : Process $\infty$   $\infty$  {lu} c)
              → Bisms $\infty$  {i} P P
BismsRef : {i : Size}{lu : LUniv}{c : Choice}
           (P : Process  $\infty$  {lu} c)
           → Bisms {i} P P
BismsRef+ : {i : Size}{lu : LUniv}{c : Choice}
            (P : Process+  $\infty$  {lu} c)
            → Bisms+ {i} P P

--@END

--@BEGIN@BismSRefinf

forceB (BismsRef $\infty$  P) = BismsRef (forceP P)

--@END

--@BEGIN@BismSRef

BismsRef (terminate x) = eqterminate
BismsRef (node P) = eqnode (BismsRef+ P)

--@END

--@BEGIN@BismSRefplus

bisim2E      (BismsRef+ P) e = e
bisimELab    (BismsRef+ P) e = refl
bisimENext   (BismsRef+ P) e = BismsRef $\infty$  (PE P e)
bisim2I      (BismsRef+ P) e = e
bisimINext   (BismsRef+ P) e = BismsRef $\infty$  (PI P e)
bisim2T      (BismsRef+ P) e = e
bisim2TEq    (BismsRef+ P) e = refl
bisim2Er     (BismsRef+ P) e = e
bisimELabr   (BismsRef+ P) e = refl
bisimENextr  (BismsRef+ P) e = BismsRef $\infty$  (PE P e)
bisim2Ir     (BismsRef+ P) e = e
bisimINextr  (BismsRef+ P) e = BismsRef $\infty$  (PI P e)
bisim2Tr     (BismsRef+ P) e = e

```



```
bisim2TEqr (BismsRef+ P) e = refl
```

```
--@END
```

A.23 bisimwImpliesStableFailuresEquivalence.agda

```
--@PREFIX@bisimwImpliesStableFailuresEquivalence
```

```
module bisimwImpliesStableFailuresEquivalence where
```

```
open import process
open import choiceSetU
open import labelUniv
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Unit.Base
open import Data.Empty
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import fdi
open import fdiRefusal
open import bisimilarity
open import bisimSym
open import Data.Bool    renaming (T to True)
open import Data.Product
open import bisimForNextProcess
open import traceImpliesTraceP
open import bisimImpliesBisim
open import bisimilarityProofs
open import auxData
open import bisimImpliesTraceEquiv
open import bisimSImpliesBisimw
```

```
--@BEGIN@bisimRefusalrosPlusStabFailRefsfOne
```



```

bisimwImplies $\sqsubseteq_{sf_1}$  : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
   $\rightarrow$  P  $\sqsubseteq_{sf_1}$  P'
bisimwImplies $\sqsubseteq_{sf_1}$  {lu}{c} P P' PP' l X
  (stableFp Q' tr' stab' drefuse')
  = (stableFp Qhat trhat2
    (
      stabSchNoTickIfRos2StablePar Qhat
      true stabSchQhat stabNoTick )
    drefusehat)
where
Qcom : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
Qcom = bisimTraceTrP1 P P' PP' l (inj1 Q') tr'

trcom : TrP {lu} l (bisimTraceTrP1 P P' PP' l
  (inj1 Q') tr') P
trcom = bisimTraceTrP2 P P' PP' l (inj1 Q') tr'

QQ'com : BisimForNextP (bisimTraceTrP1 P P' PP' l
  (inj1 Q') tr') (inj1 Q')
QQ'com = bisimTraceTrP3 P P' PP' l (inj1 Q') tr'

Q : Process  $\infty$  {lu} c
Q = lemmayyy1 Qcom Q' stab' X drefuse' QQ'com

tr : TrP {lu} l (inj1 Q) P
tr = lemmayyy2 Qcom l P Q' stab' X drefuse' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmayyy3 Qcom Q' stab' X drefuse' QQ'com

Qhat : Process  $\infty$  {lu} c
Qhat = nonDivBecomeStable1 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))

trhat : TrP {lu} [] (inj1 Qhat) Q
trhat = nonDivBecomeStable2 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))

```

```

QhatQ' :      Bisimw Qhat Q'
QhatQ' =      bisimPPWithEmptyTr Q Q' QQ' stab'
              (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
               (stableImpliesNonDiv Q' stab')) trhat

stabSchQhat : stableSch Qhat
stabSchQhat   = nonDivBecomeStable3 c Q
              (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
               (stableImpliesNonDiv Q' stab'))

stabNoTick : noTickIfRoscoe true Qhat
stabNoTick   = bisimwStableToNoTick Qhat Q' QhatQ' stab' stabSchQhat

trhat1      : TrP {lu} (l ++ []) (inj1 Qhat) P
trhat1 = trPAppendTrw c P Q l [] (inj1 Qhat) tr trhat

eql :      (l ++ []) ≡ l
eql = lemEqList l

trhat2      : TrP {lu} l (inj1 Qhat) P
trhat2      = subst (λ l' → TrP {lu} l' (inj1 Qhat) P) eql trhat1

drefusehat :      DRefusal Qhat true X
drefusehat =      bisimDRefusal Q' Qhat stab'
                  (BismwSym Qhat Q' QhatQ') X true drefuse'

--@END

--@BEGIN@bisimRefusalrosPlusStabFailRefsfOnePlus

bisimwImpliessf1+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
                  (P' : Process+ ∞ {lu} c)
                  (PP' : Bisimw+ {∞} P P')
                  → P sf1+ P'

--@END

bisimwImpliessf1+ {lu}{c} P P' PP' l X
  (stableFp Q' tr' stab' drefuse')
  = (stableFp Qhat trhat2
     (stabSchNoTickIfRos2StablePar Qhat true
      stabSchQhat stabNoTick ))

```

```

                                drefusehat)
where
  Qcom : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c
  Qcom = bisimTraceTrP1+ P P' PP' l (inj1 Q') tr'

  trcom : TrP+ l (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') P
  trcom = bisimTraceTrP2+ P P' PP' l (inj1 Q') tr'

  QQ'com : BisimForNextP
            (bisimTraceTrP1+ P P' PP' l (inj1 Q') tr') (inj1 Q')
  QQ'com = bisimTraceTrP3+ P P' PP' l (inj1 Q') tr'

  Q : Process  $\infty$  {lu} c
  Q = lemmayyy1 Qcom Q' stab' X drefuse' QQ'com

  tr : TrP+ {lu} l (inj1 Q) P
  tr = lemmayyy2+ Qcom l P Q' stab' X drefuse' QQ'com trcom

  QQ' : Bisimw Q Q'
  QQ' = lemmayyy3 Qcom Q' stab' X drefuse' QQ'com

  Qhat : Process  $\infty$  {lu} c
  Qhat = nonDivBecomeStable1 c Q
            (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
             (stableImpliesNonDiv Q' stab'))

  trhat : TrP {lu} [] (inj1 Qhat) Q
  trhat = nonDivBecomeStable2 c Q
            (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
             (stableImpliesNonDiv Q' stab'))

  QhatQ' : Bisimw Qhat Q'
  QhatQ' = bisimPPWithEmptyTr Q Q' QQ' stab'
            (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
             (stableImpliesNonDiv Q' stab')) trhat

  stabSchQhat : stableSch Qhat
  stabSchQhat = nonDivBecomeStable3 c Q
            (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
             (stableImpliesNonDiv Q' stab'))

```

```

stabNoTick : noTickIfRoscoe true Qhat
stabNoTick   = bisimwStableToNoTick Qhat Q' QhatQ' stab' stabSchQhat

trhat1      : TrP+ {lu} (l ++ []) (inj1 Qhat) P
trhat1 = trPAppendTrw+ c P Q l [] (inj1 Qhat) tr trhat

eql : (l ++ []) ≡ l
eql = lemEqList l

trhat2      : TrP+ {lu} l (inj1 Qhat) P
trhat2      = subst (λ l' → TrP+ {lu} l' (inj1 Qhat) P) eql trhat1

drefusehat :      DRefusal Qhat true X
drefusehat =      bisimDRefusal Q' Qhat stab'
                  (BismwSym Qhat Q' QhatQ') X true drefuse'

--@BEGIN@bisimRefusalrosPlusStabFailRefsfOneInf

bisimwImplies⊆sf1∞ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
                    (P' : Process∞ ∞ {lu} c)
                    (PP' : Bisimw∞ {∞} P P')
                    → P ⊆sf1∞ P'

--@END

bisimwImplies⊆sf1∞ {lu} {c} P P' PP' l X
  (stableFp Q' tr' stab' drefuse') =
    stableFp Qhat trhat2
    (stabSchNoTickIfRos2StablePar Qhat true stabSchQhat
     stabNoTick)
    drefusehat

where
  Qcom : Process ∞ {lu} c ⊔ ChoiceSet c
  Qcom = bisimTraceTrP1 (forcep P) (forcep P') (forceB PP') l (inj1 Q') tr'

  trcom : TrP l (bisimTraceTrP1 (forcep P) (forcep P') (forceB PP') l (inj1 Q') tr')
  trcom = bisimTraceTrP2 (forcep P) (forcep P') (forceB PP') l (inj1 Q') tr'

  QQ'com : BisimForNextP
  QQ'com = bisimTraceTrP3 (forcep P) (forcep P') (forceB PP') l (inj1 Q') tr'

```

```

Q : Process ∞ {lu} c
Q = lemmayyy1 Qcom Q' stab' X drefuse' QQ'com

tr : TrP {lu} l (inj1 Q) (forcep P)
tr = lemmayyy2 Qcom l (forcep P) Q' stab' X drefuse' QQ'com trcom

QQ' : Bisimw Q Q'
QQ' = lemmayyy3 Qcom Q' stab' X drefuse' QQ'com

Qhat : Process ∞ {lu} c
Qhat = nonDivBecomeStable1 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))

trhat : TrP {lu} [] (inj1 Qhat) Q
trhat = nonDivBecomeStable2 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))

QhatQ' : Bisimw Qhat Q'
QhatQ' = bisimPPWithEmptyTr Q Q' QQ' stab'
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab')) trhat

stabSchQhat : stableSch Qhat
stabSchQhat = nonDivBecomeStable3 c Q
      (bisimStableImpliesNotDivergent c Q Q' QQ' stab'
       (stableImpliesNonDiv Q' stab'))

stabNoTick : noTickIfRoscoe true Qhat
stabNoTick = bisimwStableToNoTick Qhat Q' QhatQ' stab' stabSchQhat

trhat1 : TrP {lu} (l ++ []) (inj1 Qhat) (forcep P)
trhat1 = trPAppendTrw c (forcep P) Q l [] (inj1 Qhat) tr trhat

eqI : (l ++ []) ≡ l
eqI = lemEqList l

trhat2 : TrP {lu} l (inj1 Qhat) (forcep P)
trhat2 = subst (λ l' → TrP {lu} l' (inj1 Qhat) (forcep P)) eqI trhat1

```

```

drefusehat :      DRefusal Qhat true X
drefusehat =      bisimDRefusal Q' Qhat stab'
                  (BismwSym Qhat Q' QhatQ') X true drefuse'

```

```

{- now the reversed versions-}

```

```

--@BEGIN@bisimRefusalrosPlusStabFailRefsfOneR

```

```

bisimwImplies⊆sf1r+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
                    (P' : Process+ ∞ {lu} c)
                    (PP' : Bisimw+ {∞} P P')
                    → P' ⊆sf1+ P

```

```

bisimwImplies⊆sf1r+ P P' PP' =
  bisimwImplies⊆sf1+ P' P
                                (BismwSym+ P P' PP')

```

```

--@END

```

```

bisimwImplies⊆sf1r∞ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
                    (P' : Process∞ ∞ {lu} c)
                    (PP' : Bisimw∞ {∞} P P')
                    → P' ⊆sf1∞ P

```

```

bisimwImplies⊆sf1r∞ P P' PP' = bisimwImplies⊆sf1∞ P' P (BismwSym∞ P P' PP')

```

```

bisimwImplies⊆sf1r : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
                    (P' : Process ∞ {lu} c)
                    (PP' : Bisimw {∞} P P')
                    → P' ⊆sf1 P

```

```

bisimwImplies⊆sf1r P P' PP' = bisimwImplies⊆sf1 P' P (BismwSym P P' PP')

```

```

{- proof of refinement w.r.t. ⊆sf -}

```

```

--@BEGIN@bisimRefusalrosPlusStabFailRefsf

```

```

bisimwImplies⊆sf+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
                    (P' : Process+ ∞ {lu} c)
                    (PP' : Bisimw+ {∞} P P')
                    → P ⊆sf+ P'

```

```

bisimwImplies $\sqsubseteq$ sf+ P P' PP'
  = bisimTraceEq+ P P' PP' „ bisimwImplies $\sqsubseteq$ sf1+ P P' PP'

--@END

bisimwImplies $\sqsubseteq$ sf $\infty$  : {lu : LUniv}{c : Choice} (P : Process $\infty$   $\infty$  {lu} c)
  (P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')
  → P  $\sqsubseteq$ sf $\infty$  P'
bisimwImplies $\sqsubseteq$ sf $\infty$  P P' PP' = bisimTraceEq $\infty$  P P' PP' „ bisimwImplies $\sqsubseteq$ sf1 $\infty$  P P' PP'

bisimwImplies $\sqsubseteq$ sf : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  → P  $\sqsubseteq$ sf P'
bisimwImplies $\sqsubseteq$ sf P P' PP' = bisimTraceEq P P' PP' „ bisimwImplies $\sqsubseteq$ sf1 P P' PP'

{- proof of refinement reversed w.r.t.  $\sqsubseteq$ sf -}

--@BEGIN@bisimRefusalrosPlusStabFailRefsfR

bisimwImplies $\sqsubseteq$ sfr : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
  → P'  $\sqsubseteq$ sf P
bisimwImplies $\sqsubseteq$ sfr P P' PP'
  = bisimwImplies $\sqsubseteq$ sf P' P (BismwSym P P' PP')

--@END

bisimwImplies $\sqsubseteq$ sfr+ : {lu : LUniv}{c : Choice} (P : Process+  $\infty$  {lu} c)
  (P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
  → P'  $\sqsubseteq$ sf+ P
bisimwImplies $\sqsubseteq$ sfr+ P P' PP' = bisimwImplies $\sqsubseteq$ sf+ P' P (BismwSym+ P P' PP')

bisimwImplies $\sqsubseteq$ sfr $\infty$  : {lu : LUniv}{c : Choice} (P : Process $\infty$   $\infty$  {lu} c)
  (P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')
  → P'  $\sqsubseteq$ sf $\infty$  P

```

```
bisimwImplies $\sqsubseteq$ sfr $\infty$  P P' PP' = bisimwImplies $\sqsubseteq$ sfr $\infty$  P' P (BismwSym $\infty$  P P' PP')
```

```
{- proof of equality -}
```

```
--@BEGIN@bisimRefusalrosPlusStabFailRefsEquiv
```

```
bisimwImplies=sf : {lu : LUniv}{c : Choice}
  (P : Process  $\infty$  {lu} c)
  (P' : Process  $\infty$  {lu} c)
  (PP' : Bisimw { $\infty$ } P P')
   $\rightarrow$  P =sf P'
```

```
--@END
```

```
--@BEGIN@bisimRefusalrosPlusStabFailRefsEquivProof
```

```
bisimwImplies=sf P P' PP'
  = bisimwImplies $\sqsubseteq$ sf P P' PP' „
    bisimwImplies $\sqsubseteq$ sfr P P' PP'
```

```
--@END
```

```
bisimwImplies=sf+ : {lu : LUniv}{c : Choice} (P : Process+  $\infty$  {lu} c)
  (P' : Process+  $\infty$  {lu} c)
  (PP' : Bisimw+ { $\infty$ } P P')
   $\rightarrow$  P =sf+ P'
```

```
bisimwImplies=sf+ P P' PP'
  = bisimwImplies $\sqsubseteq$ sf+ P P' PP' „
    bisimwImplies $\sqsubseteq$ sfr+ P P' PP'
```

```
bisimwImplies=sf $\infty$  : {lu : LUniv}{c : Choice} (P : Process $\infty$   $\infty$  {lu} c)
  (P' : Process $\infty$   $\infty$  {lu} c)
  (PP' : Bisimw $\infty$  { $\infty$ } P P')
   $\rightarrow$  P =sf $\infty$  P'
```

```
bisimwImplies=sf $\infty$  P P' PP' = bisimwImplies $\sqsubseteq$ sf $\infty$  P P' PP' „
  bisimwImplies $\sqsubseteq$ sfr $\infty$  P P' PP' {-bisimwImplies $\sqsubseteq$ sfr $\infty$  P P' PP' -}
```

```
{- now same, but using proofs of strong bisimilarity -}
```

```

bisimslmplies=sf+ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
                  (P' : Process+ ∞ {lu} c)
                  (PP' : Bisims+ {∞} P P')
                  → P =sf+ P'
bisimslmplies=sf+ P P' PP' = bisimwlmplies=sf+ P P' (bisimsToBismw+ P P' PP')

bisimslmplies=sf∞ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
                  (P' : Process∞ ∞ {lu} c)
                  (PP' : Bisims∞ {∞} P P')
                  → P =sf∞ P'
bisimslmplies=sf∞ P P' PP' = bisimwlmplies=sf∞ P P' (bisimsToBismw∞ P P' PP')

--@BEGIN@bisimRefusalrosPlusStabFailRefsEquivStrong

bisimslmplies=sf :    {lu : LUniv}{c : Choice}
                    (P : Process ∞ {lu} c)
                    (P' : Process ∞ {lu} c)
                    (PP' : Bisims {∞} P P')
                    → P =sf P'

--@END

--@BEGIN@bisimRefusalrosPlusStabFailRefsEquivStrongProof

bisimslmplies=sf P P' PP' =
  bisimwlmplies=sf P P' (bisimsToBismw P P' PP')

--@END

```

A.24 choiceAuxFunction.agda

```

--@PREFIX@choiceAuxFunction
module choiceAuxFunction where

```

```

open import auxData
open import choiceSetU
open import Data.Bool
open import Data.Maybe
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List.Base renaming (map to mapL)
open import Data.Sum
open import Data.Product hiding ( _×_ )

extChoiceElToName : String → String
extChoiceElToName s = "e-" ++s s

intChoiceElToName : String → String
intChoiceElToName s = "i-" ++s s

terminationChoiceElToName : String → String
terminationChoiceElToName s = s

choice2EnumWithStr : (c : Choice) → List (String × ChoiceSet c)
choice2EnumWithStr c = mapL (λ a → (choice2Str a ,, a)) (choice2Enum c)

mutual
--@BEGIN@lookupInEnum

lookupInEnum : {A : Set} → List (String × A) → String → Maybe A
lookupInEnum [] str = nothing
lookupInEnum ((str' ,, a) :: l) str = lookupInEnumAux a l str
                                   (str' ==strb str)

lookupInEnumAux : {A : Set} → A → List (String × A) → String → Bool
                                   → Maybe A
lookupInEnumAux a l s false = lookupInEnum l s
lookupInEnumAux a l s true  = just a

--@END

--@BEGIN@lookupChoice

combineEnumerations : {E I : Choice} → List (String × ChoiceSet E)
                                   → List (String × ChoiceSet I)
                                   → List (String × (ChoiceSet E ⊔ ChoiceSet I))

```

```

combineEnumerations {E} {I} L L' =
  (mapL (λ {( s ,, c)
    → (extChoiceElToName s ,, inj1 c)}) L)
  ++
  (mapL (λ {( s ,, c) → (intChoiceElToName s ,, inj2 c)}) L')

lookupChoice : (E I : Choice) → String
              → Maybe (ChoiceSet E ⊔ ChoiceSet I)
lookupChoice E I s = lookupInEnum (combineEnumerations
                                   (choice2EnumWithStr E)
                                   (choice2EnumWithStr I)) s

--@END

```

A.25 choiceFromList.agda

```

--@PREFIX@choiceFromList

module choiceFromList where

open import Size
open import process
open import choiceSetU
open import Data.List
open import primitiveProcess
open import labelUniv
open import externalChoice
open import renamingResult
open import Data.Sum
open import auxData
open import dataAuxFunction

c⊔'c→c : {c : Choice} (x : ChoiceSet (c ⊔' c)) → ChoiceSet c
c⊔'c→c (inj1 x) = x
c⊔'c→c (inj2 y) = y

--@BEGIN@choiceList

```

```

|□| : {i : Size} {c : Choice} {lu : LUniv} {A : Set} → List A
      → (A → Process i {lu} c) → Process i {lu} c
|□| {i} {c} [] f = STOP c
|□| {i} {c} (a :: []) f = f a
|□| {i} {c} (a :: (b :: l)) f = fmap c⊕'c→c ((f a) □ ((|□| (b :: l) f)))

--@END

--@BEGIN@choiceListinf

|□|∞ : {i : Size} {c : Choice} {lu : LUniv} {A : Set} → List A
      → (A → Process∞ i {lu} c) → Process∞ i {lu} c
|□|∞ {i} {c} [] f = STOP∞ c
|□|∞ {i} {c} (a :: []) f = f a
|□|∞ {i} {c} (a :: b :: l) f = fmap∞ c⊕'c→c ((f a) □∞∞ ((|□|∞ (b :: l) f)))

--@END

```

A.26 choiceSetU.agda

```

--@PREFIX@ChoU

module choiceSetU where

open import auxData
open import dataAuxFunction
open import Data.Bool
open import Data.Nat
open import Data.Fin renaming (_+_ to _+_,_<_ to _<F_)
open import Data.String renaming (_==_ to _==strb_,_++_ to _++s_)
open import Data.Nat.Show renaming (show to showℕ)
open import Data.List.Base renaming (map to mapL)
open import Data.Maybe
open import Agda.Builtin.Unit
open import Data.Product hiding ( _×_ )
open import Data.Sum

```

```

infixl 3 _⊕'_
infixl 4 _×'_

-- \ChoU
--@BEGIN@choSetU

data NamedElements (s : List String) : Set where
  ne : Fin (length s) → NamedElements s

mutual
  data Choice : Set where
    fin : ℕ → Choice
    _⊕'_ : Choice → Choice → Choice
    _×'_ : Choice → Choice → Choice
    subset' : (E : Choice) → (ChoiceSet E → Bool) → Choice
    Σ'      : (E : Choice) → (ChoiceSet E → Choice) → Choice
    namedElements : List String → Choice
    list      : (E : Choice) (l : List (ChoiceSet E)) → Choice

  ChoiceSet : Choice → Set
  ChoiceSet (fin n) = Fin n
  ChoiceSet (s ⊕' t) = ChoiceSet s ⊕ ChoiceSet t
  ChoiceSet (E ×' F) = ChoiceSet E × ChoiceSet F
  ChoiceSet (Σ' A B) = Σ[ x ∈ ChoiceSet A ] ChoiceSet (B x)
  ChoiceSet (namedElements s) = NamedElements s
  ChoiceSet (subset' E f) = subset (ChoiceSet E) f
  ChoiceSet (list E l) = ListChoiceElements E l

  data ListChoiceElements (E : Choice)(l : List (ChoiceSet E)) : Set where
    lce : Fin (length l) → ListChoiceElements E l

--@END

∅' : Choice
∅' = fin 0

T' : Choice
T' = fin 1

nth : {A : Set} → (l : List A) → Fin (length l) → A

```

```

nth [] ()
nth (a :: l) zero = a
nth (a :: l) (suc n) = nth l n

```

```
--@BEGIN@choicetwoStr
```

```

choice2Str : {c : Choice} → ChoiceSet c → String
choice2Str {fin n} m = showN (toN m)
choice2Str {c₀ ⊔' c₁} (inj₁ a) =
  "(inl " ++s (choice2Str {c₀} a) ++s ")"
choice2Str {c₀ ⊔' c₁} (inj₂ a) =
  "(inr " ++s (choice2Str {c₁} a) ++s ")"
choice2Str {c₀ ×' c₁} (x₁ , x₂) =
  "(" ++s (choice2Str {c₀} x₁) ++s
  "," ++s (choice2Str {c₁} x₂) ++s ")"
choice2Str {namedElements s} (ne i) = nth s i
choice2Str {Σ' c₀ c₁} (x₁ , x₂) =
  (choice2Str {c₀} x₁) ++s ","
  ++s (choice2Str {c₁} x₂)
choice2Str {subset' E f} (sub a x) = choice2Str {E} a
choice2Str {list E l} (lce i) = choice2Str {E} (nth l i)

```

```
--@END
```

```

choice2Stri : (c : Choice) → ChoiceSet c → String
choice2Stri c a = choice2Str {c} a

```

```

boolToMaybeTrue : (b : Bool) → Maybe (T b)
boolToMaybeTrue false = nothing
boolToMaybeTrue true = just tt

```

```

set2MaybeSubset0 : (A : Set) → (f : A → Bool) → (a : A) → Maybe (T (f a))
  → Maybe (subset A f)
set2MaybeSubset0 A f a (just p) = just (sub a p)
set2MaybeSubset0 A f a nothing = nothing

```

```
set2MaybeSubset : (A : Set) → (f : A → Bool) → A → Maybe (subset A f)
set2MaybeSubset A f a = set2MaybeSubset0 A f a (boolToMaybeTrue (f a))
```

```
--@BEGIN@choicetwoEnum
```

```
choice2Enum : (c : Choice) → List (ChoiceSet c)
choice2Enum (fin n)      = fin2Option0 n
choice2Enum (c0 ⊔' c1) = mapL (λ a → inj1 a)
                        (choice2Enum c0) ++
                        mapL (λ a → inj2 a) (choice2Enum c1)
choice2Enum (c0 ×' c1) = concat (mapL (λ a → (mapL (λ b → (a , b))
                        (choice2Enum c1)) (choice2Enum c0)))
choice2Enum (namedElements s) = mapL (λ i → ne i) (fin2Option0 (length s))
choice2Enum (Σ' c0 c1) = concat (mapL (λ a → (mapL (λ b → (a , b))
                        (choice2Enum (c1 a)) (choice2Enum c0)))
choice2Enum (subset' E f) = gfilter (set2MaybeSubset
                        (ChoiceSet E) f) (choice2Enum E)
choice2Enum (list E l) = mapL lce (fin2Option0 (length l))
```

```
--@END
```

```
choicelsEmpty : Choice → Bool
choicelsEmpty c = null (choice2Enum c)
```

```
∅ ⊔ ∅ → ∅ : {A : Set} → ChoiceSet (∅ ⊔' ∅) → A
∅ ⊔ ∅ → ∅ (inj1 ())
∅ ⊔ ∅ → ∅ (inj2 ())
```

A.27 choiceSetUOptimized.agda

```
module choiceSetUOptimized where
```

```
open import auxData
open import dataAuxFunction
```

```

open import choiceSetU
open import Data.Bool
open import Data.Nat
open import Data.Fin renaming (_+_ to _+,_<_ to _<F_)
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.Nat.Show renaming (show to showℕ)
open import Data.List.Base renaming (map to mapL)
open import Data.Maybe
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product hiding ( _×_ )
open import Data.Sum
open import Data.Empty
open import Relation.Binary.PropositionalEquality

```

mutual

```

choice2OptimizedChoiceaux⊕' : (c : Choice)(b : Bool)(c' : Choice)(b' : Bool) → Choice
choice2OptimizedChoiceaux⊕' c false c' false = (choice2OptimizedChoice c) ⊕' (choice2OptimizedChoice c')
choice2OptimizedChoiceaux⊕' c false c' true = (choice2OptimizedChoice c)
choice2OptimizedChoiceaux⊕' c true c' b' = (choice2OptimizedChoice c')

```

```

choice2OptimizedChoiceaux×' : (c : Choice)(b : Bool)(c' : Choice)(b' : Bool) → Choice
choice2OptimizedChoiceaux×' c false c' false = (choice2OptimizedChoice c) ×' (choice2OptimizedChoice c')
choice2OptimizedChoiceaux×' c false c' true = fin 0
choice2OptimizedChoiceaux×' c true c' b' = fin 0

```

```

choice2OptimizedChoiceauxSubset : (c : Choice)(f : ChoiceSet c → Bool)(b : Bool)
    → Choice
choice2OptimizedChoiceauxSubset c f false = subset' c f
choice2OptimizedChoiceauxSubset c f true = fin 0

```

```

choice2OptimizedChoice : (c : Choice) → Choice
choice2OptimizedChoice (fin x) = fin x
choice2OptimizedChoice (c ⊕' c₁) = choice2OptimizedChoiceaux⊕' c (choiceIsEmpty c) c₁ (choice2OptimizedChoice c₁)
choice2OptimizedChoice (c ×' c₁) = choice2OptimizedChoiceaux×' c (choiceIsEmpty c) c₁ (choice2OptimizedChoice c₁)
choice2OptimizedChoice (namedElements x) = namedElements x
choice2OptimizedChoice (subset' c f) = choice2OptimizedChoiceauxSubset c f (choiceIsEmpty c)
choice2OptimizedChoice (Σ' c x) = Σ' c x
choice2OptimizedChoice (list c l) = list c l

```

```

choice2OptimizedChoiceaux⊔'2choice : (c : Choice)(b : Bool)(c' : Choice)(b' : Bool)
  (x : ChoiceSet (choice2OptimizedChoiceaux⊔' c b c' b'))
  → ChoiceSet (c ⊔' c')
choice2OptimizedChoiceaux⊔'2choice c false c' false (inj1 x) = inj1 (choice2OptimizedChoice2choice c
choice2OptimizedChoiceaux⊔'2choice c false c' false (inj2 y) = inj2 (choice2OptimizedChoice2choice c
choice2OptimizedChoiceaux⊔'2choice c false c' true x = inj1 (choice2OptimizedChoice2choice c x)
choice2OptimizedChoiceaux⊔'2choice c true c' b' y = inj2 (choice2OptimizedChoice2choice c' y)

choice2OptimizedChoiceaux×'2choice : (c : Choice)(b : Bool)(c' : Choice)(b' : Bool)
  (x : ChoiceSet (choice2OptimizedChoiceaux×' c b c' b'))
  → ChoiceSet (c ×' c')
choice2OptimizedChoiceaux×'2choice c false c' false (x ,, x') =
  choice2OptimizedChoice2choice c x ,, choice2OptimizedChoice2choice c' x'
choice2OptimizedChoiceaux×'2choice c false c' true ()
choice2OptimizedChoiceaux×'2choice c true c' b' ()

choice2OptimizedChoiceauxSubset2choice : (c : Choice)(f : ChoiceSet c → Bool)(b : Bool)
  (x : ChoiceSet (choice2OptimizedChoiceauxSubset c f b))
  → ChoiceSet (subset' c f)
choice2OptimizedChoiceauxSubset2choice c f false x = x
choice2OptimizedChoiceauxSubset2choice c f true ()

choice2OptimizedChoice2choice : (c : Choice)(x : ChoiceSet (choice2OptimizedChoice c))
  → ChoiceSet c
choice2OptimizedChoice2choice (fin x) x1 = x1
choice2OptimizedChoice2choice (c ⊔' c1) x = choice2OptimizedChoiceaux⊔'2choice c (choiceIsEmpty
  (choiceIsEmpty c1) x)
choice2OptimizedChoice2choice (c ×' c1) x = choice2OptimizedChoiceaux×'2choice c (choiceIsEmpty
  (choiceIsEmpty c1) x)
choice2OptimizedChoice2choice (namedElements x) x1 = x1
choice2OptimizedChoice2choice (subset' c f) y = choice2OptimizedChoiceauxSubset2choice c f (choice
choice2OptimizedChoice2choice (Σ' c x) x1 = x1
choice2OptimizedChoice2choice (list c l) x1 = x1

```

A.28 choiceSetUOptimized2.agda

```
module choiceSetUOptimized2 where
```

```

open import auxData
open import dataAuxFunction
open import choiceSetU
open import Data.Bool
open import Data.Nat
open import Data.Fin renaming (_+_ to _+_,_<_ to _<F_)
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.Nat.Show renaming (show to showℕ)
open import Data.List.Base renaming (map to mapL)
open import Data.Maybe
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product hiding ( _×_ )
open import Data.Sum
open import Data.Empty
open import Relation.Binary.PropositionalEquality

```

```

choice2OptimizedChoice : (c : Choice) → Choice
choice2OptimizedChoice c = list c (choice2Enum c)

```

```

choice2OptimizedChoice2choice : (c : Choice)
                                (x : ChoiceSet (choice2OptimizedChoice c))
                                → ChoiceSet c
choice2OptimizedChoice2choice c (lce i) = nth (choice2Enum c) i

```

A.29 choiceSetUOptimized3.agda

```

module choiceSetUOptimized3 where

```

```

open import auxData
open import dataAuxFunction
open import choiceSetU
open import Data.Bool
open import Data.Nat
open import Data.Fin renaming (_+_ to _+_,_<_ to _<F_)
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.Nat.Show renaming (show to showℕ)
open import Data.List.Base renaming (map to mapL)

```

```

open import Data.Maybe
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product hiding ( _×_ )
open import Data.Sum
open import Data.Empty
open import Relation.Binary.PropositionalEquality

choice2OptimizedChoice : (c : Choice) → Choice
choice2OptimizedChoice c = fin (length (choice2Enum c))

choice2OptimizedChoice2choice : (c : Choice)
                                (x : ChoiceSet (choice2OptimizedChoice c))
                                → ChoiceSet c
choice2OptimizedChoice2choice c i = nth (choice2Enum c) i

```

A.30 dataAuxFunction.agda

```

--@PREFIX@dataauxfunction

module dataAuxFunction where

open import auxData
open import Data.Bool
open import Data.Nat
open import Agda.Builtin.Nat renaming (_<_ to _<N_; _==_ to _==N_)
open import Data.Fin renaming (_+_ to _+,_<_ to _<F_)
open import Data.Char renaming (_==_ to _==?_)
open import Data.Maybe
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.Nat.Show renaming (show to showN)
open import Data.List.Base renaming (map to mapL)
open import Data.Sum
open import Agda.Builtin.Unit
open import Data.Empty
open import Data.Product hiding ( _×_ )

¬ : Set → Set

```

```
¬ a = a → ⊥
```

```
¬b : Bool → Bool
¬b true = false
¬b false = true
```

```
tertiumNonDatur : (b : Bool) → T b ⊔ T (¬b b)
tertiumNonDatur true = inj₁ tt
tertiumNonDatur false = inj₂ tt
```

```
transfer : {A : Set} → (C : A → Set) → (a a' : A) → a == a' → C a → C a'
transfer C a .a refl c = c
```

```
projSubset : {A : Set} → {f : A → Bool} → subset A f → A
projSubset (sub a x) = a
```

```
_o_ : {A B C : Set} → (B → C) → (A → B) → A → C
(f o g) a = f (g a)
```

```
infixr 9 _o_
```

```
π₀ : {A B : Set} → A × B → A
π₀ (a , b) = a
```

```
π₁ : {A B : Set} → A × B → B
π₁ (a , b) = b
```

```
efq : {A : Set} → Fin 0 → A
efq ()
```

```
isDigit : Char → Maybe ℕ
isDigit '0' = just 0
isDigit '1' = just 1
isDigit '2' = just 2
```

```

isDigit '3' = just 3
isDigit '4' = just 4
isDigit '5' = just 5
isDigit '6' = just 6
isDigit '7' = just 7
isDigit '8' = just 8
isDigit '9' = just 9
isDigit _  = nothing

```

```

attach : Maybe ℕ → ℕ → ℕ
attach nothing n = n
attach (just m) n = 10 * m + n

```

```

Quote : List Char → Maybe (List ℕ)
Quote xs = parse xs nothing []
  where
    parse : List Char → Maybe ℕ → List ℕ → Maybe (List ℕ)
    parse [] nothing ns = just ns
    parse [] (just n) ns = just (n :: ns)
    parse (hd :: tl) m ns with isDigit hd
    ... | nothing = nothing
    ... | just n  = parse tl (just (attach m n)) ns

```

```

stringListToℕ : String → Maybe (List ℕ)
stringListToℕ xs with Quote (toList xs)
... | nothing = nothing
... | just ns = just (reverse ns)

```

```

listℕtoℕ' : List ℕ → ℕ
listℕtoℕ' [] = 0
listℕtoℕ' (n :: l) = listℕtoℕ' l * 10 + n

```

```

listℕtoℕ : List ℕ → ℕ
listℕtoℕ l = listℕtoℕ' (reverse l)

```

```

stringToMaybeℕ : String → Maybe ℕ
stringToMaybeℕ s with (stringListToℕ s)
stringToMaybeℕ s | just l = just (listℕtoℕ l)
stringToMaybeℕ s | nothing = nothing

```

```

<NboolTo< : {n m : ℕ} → T (n <N m) → n < m
<NboolTo< {zero} {zero} ()
<NboolTo< {zero} {suc m} p = s≤s z≤n
<NboolTo< {suc n} {zero} ()
<NboolTo< {suc n} {suc m} p = s≤s (<NboolTo< {n} {m} p)

sumFin : (n : ℕ) → (Fin n → ℕ) → ℕ
sumFin zero _ = 0
sumFin (suc n) f = f zero + sumFin n (f ∘ suc)

prodFin : (n : ℕ) → (Fin n → ℕ) → ℕ
prodFin zero _ = 1
prodFin (suc n) f = f zero * sumFin n (f ∘ suc)

embed : {n : ℕ} → Fin n → Fin (suc n)
embed zero = zero
embed (suc m) = suc (embed m)

last : {n : ℕ} → Fin (suc n)
last {zero} = zero
last {suc n} = suc (last {n})

fin2OptionAux : {n : ℕ} → String × Fin n → String × Fin (suc n)
fin2OptionAux (str ,, k) = (str ,, embed k)

fin2Option' : (n : ℕ) → List (String × Fin n)
fin2Option' zero = []
fin2Option' (suc n) = (showℕ n ,, last) :: mapL fin2OptionAux (fin2Option' n)

fin2Option : (n : ℕ) → List (String × Fin n)
fin2Option n = reverse (fin2Option' n)

--@BEGIN@finoption

fin2Option0' : (n : ℕ) → List (Fin n)
fin2Option0' zero = []
fin2Option0' (suc n) = last :: mapL embed (fin2Option0' n)

```

```

fin2Option0 : (n : ℕ) → List (Fin n)
fin2Option0 n = reverse (fin2Option0' n)

```

```
--@END
```

```

test = fin2Option0 5
test' = fin2Option0' 5

```

A.31 div.agda

```
--@PREFIX@div
```

```
module div where
```

```

open import Size
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List
open import process
open import auxData
open import dataAuxFunction
open import choiceSetU
open import labelUniv

```

```
--@BEGIN@DivDef
```

```
mutual
```

```

DIV∞ : {i : Size}{lu : LUniv} → {c : Choice} → Process∞ i {lu} c
forcep DIV∞ = DIV
Str∞ DIV∞ = "DIV"

```

```

DIV : {i : Size}{lu : LUniv} → {c : Choice} → Process i {lu} c
DIV = node DIV+

```

```

DIV+ : {i : Size}{lu : LUniv} → {c : Choice} → Process+ i {lu} c
DIV+ = (process+ ∅' efq efq T' (λ _ → DIV∞) ∅' efq "DIV")

```

```
--@END
```

A.32 efq.agda

```
--@PREFIX@efq
```

```
module efq where
```

```
open import Data.Fin
```

```
--@BEGIN@efqDef
```

```
efq : {A : Set} → Fin 0 → A
efq ()
```

```
--@END
```

A.33 example.agda

```
module example where
```

```
open import Data.Bool hiding (==?)
open import Data.Bool.Base renaming (T to T') hiding (==?)
open import libBool
open import libList
open import Data.String renaming (== to ==strb_; ++ to ++s_)
open import Data.List renaming (++ to ++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import UnitModule
--open import simulator
open import NativeIO
```

```

open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin
open import Data.Nat hiding (_=?_)
open import parallelSimple
open import Interleave
open import hidingOperator
-- open import SizedIO.Base renaming (force to forceIO; delay to delayIO)

{- needed possibly for compilation -}
--open import SizedIO.Base
open import renamingResult
open import dataAuxFunction
open import Relation.Nullary.Decidable
open import Data.String
open import simulatorCutDown hiding (main;transition1;transition2)
open import primitiveProcess
open import choiceAuxFunction
open import Data.Sum
open import label renaming (Label to LabelSimple)
open import UnitModule

data SIGNAL : Set where
  sig1  : SIGNAL -- sig2

_==sig_ : SIGNAL → SIGNAL → Bool
_ ==sig _ = true
-- sig1 ==sig sig1 = true
-- sig2 ==sig sig2 = true
-- _ ==sig _ = false

refl==sig : {s : SIGNAL} → T' (s ==sig s)
refl==sig {sig1} = _
-- refl==sig {sig2} = _

showSIGNAL : (s : SIGNAL) → String
showSIGNAL _ = "" -- "sig1"
-- showSIGNAL sig2 = "sig2"

```

LabelListSignal : List SIGNAL

LabelListSignal = [] -- sig1 :: sig1 :: []

symSignal : {l l' : SIGNAL} → T' (l ==sig l') → T' (l' ==sig l)

symSignal {sig1} {sig1} _ = _

transfSignal : (l l' : SIGNAL) (Q : SIGNAL → Set) → T' (l ==sig l') → Q l → Q l'

transfSignal sig1 sig1 Q _ x = x

open LUniv

labelSignal1 : LUniv

labelSignal1 = record{ Labelf = SIGNAL ;
 ==lf = λ s s' → s ==sig s' ;
 refl==lf = λ {l} → refl==sig {l} ;
 sym==lf = λ {s} {s'} → symSignal {s} {s'} ;
 transf = λ {s} {s'} → transfSignal s s' ;
 showLabelf = showSignal ;
 LabelListf = LabelListSignal }

-- LUniv.Labelf labelSignal1 = SIGNAL

-- LUniv._==lf_ labelSignal1 s s' = debug (s ==sig s') "_==sig "

-- LUniv.refl==lf labelSignal1 {l} = refl==sig {l}

-- LUniv.showLabelf labelSignal1 = showSignal

-- LUniv.LabelListf labelSignal1 = LabelListSignal

labelSignal : LUniv

labelSignal = labelSignal1

{- start hidden -}

transition₁ : ∀ i → Process i {lsimple} setSTOP

transition₁ i = (lab laba) → delay ((lab labb) → delay ((lab labc) → delay (STOP {∞})))

transition₂ : ∀ i → Process i {lsimple} setSTOP

transition₂ i = (lab laba) → delay ((lab labb) → delay ((lab labc) → delay {↑ (↑ i)} (STOP {∞})))

```

transition1+ : Process+ ∞ {labelSIGNAL} setSTOP
transition1+ = -- lab sig1 → delay {∞} (lab sig1 → delay {↑ ∞}
              lab sig1 →+ delay {↑ (↑ ∞)}) (STOP setSTOP)

```

```

main' : NativeIO Unit
main' = translateIOConsole (simulator (transition1 ∞))

```

```

Str' : {i : Size} → {c : Choice} → {lu : LUniv} → Process i {lu} c → String
Str' (terminate a) = "test1" -- "terminate(++s choice2Str a ++s)"
Str' (node P)      = Str+ P

```

```

record testRec : Set where
  field
    a : ℕ

```

```

open testRec

```

```

x : testRec
a x = 3

```

```

y : ℕ
y = a x

```

```

z = Labelf labelSIGNAL1

```

```

mutual
  record LUniv' : Set1 where
    constructor luniv'
    inductive
    field
      Labelf : Set
      showLabelf' : Labelf → String
      eqLf : Bool -- Labelf → Bool

```

```

data Label' (lu : LUniv') : Set where
  lab : LUniv'.Labelf lu → Label' lu

```

```
open LUniv'
```

```
const'' : SIGNAL → Bool
const''  = λ l → true
```

```
showSIGNAL' : SIGNAL → String
showSIGNAL' s = "signal"
```

```
labelSIGNAL1' : LUniv'
labelSIGNAL1' = record{ Labelf = SIGNAL ;
                        showLabelf' = showSIGNAL' ;
                        eqlf = true -- const'' -- λ l → true;
                      }
```

```
showLabel' : {lu : LUniv} → Label lu → String
showLabel' {lu} (lab x) = LUniv.showLabelf lu x -- showSIGNAL sig1 -- -- showLabelSimple
-- showLabel' {lu} (lab x) = LUniv.showLabelf lu x -- showLabelSimple
```

```
showLabel'' : {lu : LUniv'} → Label' lu → String
showLabel'' {lu} (lab x) = LUniv'.showLabelf' lu x
```

```
-- works
main'' : NativeIO Unit
main'' = nativePutStrLn (showSIGNAL sig1)
        native»= (λ r → nativeReturn unit)
```

```
-- works
main3 : NativeIO Unit
main3 = nativePutStrLn (LUniv'.showLabelf' labelSIGNAL1' sig1)
```

```
-- doesn't work
main4 : NativeIO Unit
main4 = nativePutStrLn (LUniv.showLabelf labelSIGNAL1 sig1)
        native»= (λ r → nativeReturn unit)
```

```
main = main4
```

```

-- transition2 :  ∀ i → Process i {labelSIGNAL}setSTOP
-- transition2 i = lab sig2 → delay {i} (lab sig2 → delay {↑ i} (lab sig2 → d
--
--
-- myResultType :  Choice
-- myResultType = setSTOP ×' setSTOP
--
-- myProcess :  Process ∞ {labelSIGNAL} myResultType
-- myProcess = transition1 ∞ ||| transition2 ∞
--
-- {- end hidden -}
--
-- main :  NativeIO Unit
-- main = translateIOConsole (simulator myProcess)

```

A.34 example2.agda

```

module example2 where

```

```

open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)

```

```

record Unit : Set where
  constructor unit

```

```

{-# COMPILER GHC Unit = data () (() #-}

```

```

{-# FOREIGN GHC import qualified Data.Text #-}
{-# FOREIGN GHC import qualified System.Environment #-}

```

```

postulate

```

```

  NativeIO    : Set → Set
  nativeReturn : {A : Set} → A → NativeIO A
  _native»=_ : {A B : Set} → NativeIO A → (A → NativeIO B) → NativeIO B

```

```

{-# BUILTIN IO NativeIO #-}

```

```
{-# COMPILE GHC NativeIO = type IO #-}
{-# COMPILE GHC nativeReturn = (\_ -> return :: a -> IO a) #-}
{-# COMPILE GHC _native»=_ = (\_ _ -> (»=) :: IO a -> (a -> IO b) -> IO b) #-}
```

```
postulate
```

```
  nativePutStrLn : String → NativeIO Unit
```

```
{-# COMPILE GHC nativePutStrLn = (\ s -> putStrLn (Data.Text.unpack s)) #-}
```

```
data Bool : Set where
```

```
  true false : Bool
```

```
-- mutual is the problem
-- if mutual is removed compilation works
-- otherwise compilation doesn't work
```

```
mutual
```

```
  record LUniv : Set1 where
```

```
    field
```

```
      Labelf : Set
```

```
      _==If_ : Labelf → Labelf → Bool
```

```
      showLabelf : Labelf → String
```

```
      LabelListf : Bool
```

```
open LUniv
```

```
mutual
```

```
  data Label (lu : LUniv) : Set where
```

```
    lab : Labelf lu → Label lu
```

```
data SIGNAL : Set where
```

```
  sig1 : SIGNAL -- sig2
```

```
_==sig_ : SIGNAL → SIGNAL → Bool
```

```
_ ==sig _ = true
```

```
showSIGNAL : (s : SIGNAL) → String
showSIGNAL _ = "" -- "sig1"
```

```
labelSIGNAL : LUniv
labelSIGNAL = record{ Labelf = SIGNAL ;
  _==lf_ = λ s s' → s ==sig s' ;
  showLabelf = showSIGNAL ;
  LabelListf = true } -- LabelListSIGNAL}
```

```
showLabel' : {lu : LUniv} → Label lu → String
showLabel' {lu} (lab x) = LUniv.showLabelf lu x
```

```
main : NativeIO Unit
main = nativePutStrLn (LUniv.showLabelf labelSIGNAL sig1)
      native»= (λ r → nativeReturn unit)
```

A.35 example3.agda

```
module example3 where
```

```
open import Data.String
open import UnitModule
open import NativeIO
```

```
data Bool : Set where
  true false : Bool
```

```
mutual
  record L : Set1 where
    inductive
    field
```



```

a : Set
b : a → String
c : Bool

open L

l : L
a l = Bool
b l = λ x → "test"
c l = true

main : NativeIO Unit
main = nativePutStrLn (b l true) native»= λ r → nativeReturn _

```

A.36 externalChoice.agda

```

--@PREFIX@ExternalChoice

module externalChoice where

open import Size
open import process
open import choiceSetU
open import choiceAuxFunction
open import dataAuxFunction
open import auxData
open import renamingResult
open import Size
open import Data.String renaming (_++_ to _++s_)
open import showFunction
open import Data.Sum
open import addTick
open import labelUniv

_□Res_ : Choice → Choice → Choice
c₀ □Res c₁ = (c₀ ⊕' c₁) ⊕' (c₀ ×' c₁)

```



```
--@BEGIN@ExtCDef
```

```
_□Str_ : String → String → String
s □Str s' = "(" ++ s ++ s' □ " " ++ s' ++ s ++ ")"
```

```
mutual
```

```
_□∞∞_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
      → Process∞ i {lu} c0 → Process∞ i {lu} c1
      → Process∞ i {lu} (c0 ⊕' c1)
forcep (P □∞∞ Q) = forcep P □ forcep Q
Str∞ (P □∞∞ Q) = Str∞ P □Str Str∞ Q
```

```
_□∞+_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
      → Process∞ i {lu} c0 → Process+ i {lu} c1
      → Process∞ i {lu} (c0 ⊕' c1)
forcep (P □∞+ Q) = forcep P □p+ Q
Str∞ (P □∞+ Q) = Str∞ P □Str Str+ Q
```

```
_□+∞_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
      → Process+ i {lu} c0 → Process∞ i {lu} c1
      → Process∞ i {lu} (c0 ⊕' c1)
forcep (P □+∞ Q) = P □+p forcep Q
Str∞ (P □+∞ Q) = Str+ P □Str Str∞ Q
```

```
_□_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size} → Process i {lu} c0
      → Process i {lu} c1 → Process i {lu} (c0 ⊕' c1)
node P □ Q = P □+p Q
P □ node Q = P □p+ Q
terminate a □ terminate b = 2-✓ a b
```

```
_□+p_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
      → Process+ i {lu} c0 → Process i {lu} c1
      → Process i {lu} (c0 ⊕' c1)
P □+p terminate b = addTimed✓ (inj2 b) (node (fmap+ inj1 P) )
P □+p node Q = node (P □+ Q)
```

```

_□+∞+_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
        → Process+ i {lu} c0 → Process∞ i {lu} c1
        → Process∞ i {lu} (c0 ⊕' c1)
forcep (P    □+∞+ Q) = node (P □+p+ forcep Q)
Str∞   (P    □+∞+ Q) = Str+ P □Str   Str∞ Q

```

```

_□p+_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size} → Process i {lu} c0
        → Process+ i {lu} c1 → Process i {lu} (c0 ⊕' c1)
terminate a □p+      Q = addTimed✓ (inj1 a)
                        (node (fmap+ inj2 Q) )
node P    □p+      Q = node (P □+ Q)

```

```

_□+_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
        → Process+ i {lu} c0 → Process+ i {lu} c1
        → Process+ i {lu} (c0 ⊕' c1)
E      (P □+ Q)      = E P ⊕' E Q
Lab    (P □+ Q) (inj1 x) = Lab P x
Lab    (P □+ Q) (inj2 x) = Lab Q x
PE     (P □+ Q) (inj1 x) = fmap∞ inj1 (PE P x)
PE     (P □+ Q) (inj2 x) = fmap∞ inj2 (PE Q x)
I      (P □+ Q)      = I P ⊕' I Q
PI     (P □+ Q) (inj1 c) = PI P c □∞+ Q
PI     (P □+ Q) (inj2 c) = P    □+∞ PI Q c
T      (P □+ Q)      = T P ⊕' T Q
PT     (P □+ Q) (inj1 c) = inj1 (PT P c)
PT     (P □+ Q) (inj2 c) = inj2 (PT Q c)
Str+   (P □+ Q)      = Str+ P □Str Str+ Q

```

```

_□∞p_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
        → Process∞ i {lu} c0 → Process i {lu} c1
        → Process∞ i {lu} (c0 ⊕' c1)
forcep (P    □∞p Q) = forcep P □    Q
Str∞   (P    □∞p Q) = Str∞ P □Str Str Q

```

```

_□p∞_ : {lu : LUniv}{c0 c1 : Choice} → {i : Size}
        → Process i {lu} c0 → Process∞ i {lu} c1

```

$$\begin{aligned}
 & \rightarrow \text{Process} \infty i \{lu\} (c_0 \uplus' c_1) \\
 \text{forcep } (P \quad \square p \infty Q) &= P \quad \square \text{forcep } Q \\
 \text{Str} \infty (P \quad \square p \infty Q) &= \text{Str } P \quad \square \text{Str Str} \infty Q
 \end{aligned}$$

--@END

$$\begin{aligned}
 _ \square + p + _ &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\
 &\rightarrow \text{Process} + i \{lu\} c_0 \rightarrow \text{Process } i \{lu\} c_1 \\
 &\rightarrow \text{Process} + i \{lu\} (c_0 \uplus' c_1) \\
 P \square + p + \text{terminate } b &= \text{addTimed} \checkmark + (\text{inj}_2 b) (\text{fmap} + \text{inj}_1 P) \\
 P \square + p + \text{node } Q &= (P \square ++ Q)
 \end{aligned}$$

$$\begin{aligned}
 _ \square p + p _ &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \rightarrow \text{Process } i \{lu\} c_0 \\
 &\rightarrow \text{Process} + i \{lu\} c_1 \rightarrow \text{Process } i \{lu\} (c_0 \uplus' c_1) \\
 \text{terminate } a \square p + p \quad Q &= \text{addTimed} \checkmark (\text{inj}_1 a) \\
 &\quad (\text{node } (\text{fmap} + \text{inj}_2 Q)) \\
 \text{node } P \quad \square p + p \quad Q &= \text{node } (P \square ++ Q)
 \end{aligned}$$

$$\begin{aligned}
 _ \square p ++ _ &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \rightarrow \text{Process } i \{lu\} c_0 \\
 &\rightarrow \text{Process} + i \{lu\} c_1 \rightarrow \text{Process} + i \{lu\} (c_0 \uplus' c_1) \\
 \text{terminate } a \square p ++ \quad Q &= \text{addTimed} \checkmark + (\text{inj}_1 a) \\
 &\quad ((\text{fmap} + \text{inj}_2 Q)) \\
 \text{node } P \quad \square p ++ \quad Q &= (P \square ++ Q)
 \end{aligned}$$

$$\begin{aligned}
 _ \square ++ _ &: \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\
 &\rightarrow \text{Process} + i \{lu\} c_0 \rightarrow \text{Process} + i \{lu\} c_1 \\
 &\rightarrow \text{Process} + i \{lu\} (c_0 \uplus' c_1) \\
 E \quad (P \square ++ Q) &= E P \uplus' E Q \\
 \text{Lab} \quad (P \square ++ Q) (\text{inj}_1 x) &= \text{Lab } P x \\
 \text{Lab} \quad (P \square ++ Q) (\text{inj}_2 x) &= \text{Lab } Q x \\
 PE \quad (P \square ++ Q) (\text{inj}_1 x) &= \text{fmap} \infty \text{inj}_1 (PE P x) \\
 PE \quad (P \square ++ Q) (\text{inj}_2 x) &= \text{fmap} \infty \text{inj}_2 (PE Q x) \\
 I \quad (P \square ++ Q) &= I P \uplus' I Q \\
 PI \quad (P \square ++ Q) (\text{inj}_1 c) &= PI P c \square \infty ++ Q \\
 PI \quad (P \square ++ Q) (\text{inj}_2 c) &= P \quad \square ++ \infty + PI Q c
 \end{aligned}$$

$$\begin{aligned} T & (P \square++ Q) &= T P \uplus' T Q \\ PT & (P \square++ Q) (\text{inj}_1 c) &= \text{inj}_1 (PT P c) \\ PT & (P \square++ Q) (\text{inj}_2 c) &= \text{inj}_2 (PT Q c) \\ \text{Str}+ & (P \square++ Q) &= \text{Str}+ P \square \text{Str} \text{Str}+ Q \end{aligned}$$

$$\begin{aligned} _ \square \infty ++ _ & : \{lu : LUniv\} \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\ & \rightarrow \text{Process} \infty i \{lu\} c_0 \rightarrow \text{Process}+ i \{lu\} c_1 \\ & \rightarrow \text{Process} \infty i \{lu\} (c_0 \uplus' c_1) \\ \text{forcep} (P \square \infty ++ Q) &= \text{node} (\text{forcep} P \square p++ Q) \\ \text{Str} \infty (P \square \infty ++ Q) &= \text{Str} \infty P \square \text{Str} \text{Str}+ Q \end{aligned}$$

mutual

$$\begin{aligned} _ \square w\text{Nam} \infty \infty _ \text{Using} _, _, _ & : \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \\ & \rightarrow \{lu : LUniv\} \\ & \rightarrow \text{Process} \infty i \{lu\} c_0 \\ & \rightarrow \text{Process} \infty i \{lu\} c_1 \\ & \rightarrow (\square name : \text{String} \rightarrow \text{String} \rightarrow \text{String}) \\ & \rightarrow (\square f\text{mapLeftName} : \text{String} \rightarrow \text{String}) \\ & \rightarrow (\square f\text{mapRightName} : \text{String} \rightarrow \text{String}) \\ & \rightarrow \text{Process} \infty i \{lu\} (c_0 \uplus' c_1) \end{aligned}$$

$$\begin{aligned} \text{forcep} (P \square w\text{Nam} \infty \infty Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName}) \\ &= \text{forcep} P \square w\text{Nam} \text{forcep} Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName} \\ \text{Str} \infty (P \square w\text{Nam} \infty \infty Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName}) &= \square name \end{aligned}$$

$$\begin{aligned} _ \square w\text{Nam} \infty + _ \text{Using} _, _, _ & : \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \rightarrow \{lu : LUniv\} \\ & \rightarrow \text{Process} \infty i \{lu\} c_0 \\ & \rightarrow \text{Process}+ i \{lu\} c_1 \\ & \rightarrow (\square name : \text{String} \rightarrow \text{String} \rightarrow \text{String}) \\ & \rightarrow (\square f\text{mapLeftName} : \text{String} \rightarrow \text{String}) \\ & \rightarrow (\square f\text{mapRightName} : \text{String} \rightarrow \text{String}) \\ & \rightarrow \text{Process} \infty i \{lu\} (c_0 \uplus' c_1) \end{aligned}$$

$$\begin{aligned} \text{forcep} (P \square w\text{Nam} \infty + Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName}) \\ &= \text{forcep} P \square w\text{Nam}+ Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName} \\ \text{Str} \infty (P \square w\text{Nam} \infty + Q \text{Using} \square name, \square f\text{mapLeftName}, \square f\text{mapRightName}) &= \square name \end{aligned}$$

$$\begin{aligned} _ \square w\text{Nam} \infty ++ _ \text{Using} _, _, _ & : \{c_0 c_1 : \text{Choice}\} \rightarrow \{i : \text{Size}\} \rightarrow \{lu : LUniv\} \\ & \rightarrow \text{Process} \infty i \{lu\} c_0 \\ & \rightarrow \text{Process}+ i \{lu\} c_1 \\ & \rightarrow (\square name : \text{String} \rightarrow \text{String} \rightarrow \text{String}) \end{aligned}$$

```

    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process∞ i {lu} (c₀ ⊕' c₁)
forcep (P □wNam∞++ Q Using □name , □fmapLeftName , □fmapRightName)
    = node (forcep P □wNamp++ Q Using □name , □fmapLeftName , □fmapRightName)
Str∞ (P □wNam∞++ Q Using □name , □fmapLeftName , □fmapRightName) = □name (Str∞

```

```

_□wNam+∞_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
    → Process+ i {lu} c₀
    → Process∞ i {lu} c₁
    → (□name : String → String → String)
    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process∞ i {lu} (c₀ ⊕' c₁)
forcep (P □wNam+∞ Q Using □name , □fmapLeftName , □fmapRightName)
    = P □wNam+p forcep Q Using □name , □fmapLeftName , □fmapRightName
Str∞ (P □wNam+∞ Q Using □name , □fmapLeftName , □fmapRightName)
    = □name (Str+ P) (Str∞ Q)

```

```
--@BEGIN@extchoiceWName
```

```

_□wNam_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
    → Process i {lu} c₀
    → Process i {lu} c₁
    → (□name : String → String → String)
    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process i {lu} (c₀ ⊕' c₁)

```

```
--@END
```

```

node P □wNam Q Using □name , □fmapLeftName , □fmapRightName
    = P □wNam+p Q Using □name , □fmapLeftName , □fmapRightName
P □wNam node Q Using □name , □fmapLeftName , □fmapRightName =
    P □wNamp+ Q Using □name , □fmapLeftName , □fmapRightName
terminate a □wNam terminate b Using □name , □fmapLeftName , □fmapRightName = 2-✓ a

```

```

_□wNamp_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size}
    → {lu : LUniv}
    → Process i {lu} c₀
    → Process i {lu} c₁
    → (□name : String → String → String)

```

```

    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process+ i {lu} (c₀ ⊕' c₁)
node P    □wNamp Q    Using □name , □fmapLeftName , □fmapRightName
  = P □wNam+p+ Q Using □name , □fmapLeftName , □fmapRightName
P    □wNamp node Q    Using □name , □fmapLeftName , □fmapRightName
  = P □wNamp++ Q Using □name , □fmapLeftName , □fmapRightName
terminate a □wNamp terminate b Using □name , □fmapLeftName , □fmapRightName
  = 2-✓+ a b

_□wNam+p_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size}
  → {lu : LUniv}
  → Process+ i {lu} c₀
  → Process i {lu} c₁
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)
  → (□fmapRightName : String → String)
  → Process i {lu} (c₀ ⊕' c₁)
P □wNam+p terminate b    Using □name , □fmapLeftName , □fmapRightName
  = addTimed✓ (inj₂ b) (node(fmapWithName+ □fmapRightName inj₁ P))
P □wNam+p node Q    Using □name , □fmapLeftName , □fmapRightName
  = node (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)
_□wNam+p+_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size}
  → {lu : LUniv}
  → Process+ i {lu} c₀
  → Process i {lu} c₁
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)
  → (□fmapRightName : String → String)
  → Process+ i {lu} (c₀ ⊕' c₁)
P □wNam+p+ terminate b Using □name , □fmapLeftName , □fmapRightName
  = addTimed✓+ (inj₂ b) (fmapWithName+ □fmapRightName inj₁ P)
P □wNam+p+ node Q    Using □name , □fmapLeftName , □fmapRightName
  = P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName

_□wNam+∞+_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
  → Process+ i {lu} c₀
  → Process∞ i {lu} c₁
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)

```

```

→ (□fmapRightName : String → String)
→ Process∞ i {lu} (c₀ ⊕' c₁)
forcep (P   □wNam+∞+ Q Using □name , □fmapLeftName , □fmapRightName)
= node (P   □wNam+p+ forcep Q Using □name , □fmapLeftName , □fmapRightName)
Str∞ (P   □wNam+∞+ Q Using □name , □fmapLeftName , □fmapRightName) = □name (Str

```

```

_□wNamp+_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
→ Process i {lu} c₀
→ Process+ i {lu} c₁
→ (□name : String → String → String)
→ (□fmapLeftName : String → String)
→ (□fmapRightName : String → String)
→ Process i {lu} (c₀ ⊕' c₁)
terminate a □wNamp+ Q Using □name , □fmapLeftName , □fmapRightName
= addTimed✓ (inj₁ a) (node (fmapWithName+ □fmapLeftName inj₂ Q) )
node P   □wNamp+ Q Using □name , □fmapLeftName , □fmapRightName
= node (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)

```

```

_□wNamp+p_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
→ Process i {lu} c₀
→ Process+ i {lu} c₁
→ (□name : String → String → String)
→ (□fmapLeftName : String → String)
→ (□fmapRightName : String → String)
→ Process i {lu} (c₀ ⊕' c₁)
terminate a □wNamp+p Q Using □name , □fmapLeftName , □fmapRightName
= addTimed✓ (inj₁ a) (node (fmapWithName+ □fmapLeftName inj₂ Q) )
node P   □wNamp+p Q Using □name , □fmapLeftName , □fmapRightName
= node (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName)

```

```

_□wNamp++_Using_,_,_ : {c₀ c₁ : Choice} → {i : Size} → {lu : LUniv}
→ Process i {lu} c₀
→ Process+ i {lu} c₁
→ (□name : String → String → String)
→ (□fmapLeftName : String → String)
→ (□fmapRightName : String → String)
→ Process+ i {lu} (c₀ ⊕' c₁)
terminate a □wNamp++ Q Using □name , □fmapLeftName , □fmapRightName

```

```

      = addTimed✓+ (inj1 a) ((fmapWithName+ □fmapLeftName inj2 Q) )
node P    □wNamp++ Q Using □name , □fmapLeftName , □fmapRightName
      = P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName

```

```

_□wNam+_Using_,_,_ : {c0 c1 : Choice} → {i : Size} → {lu : LUniv}
  → Process+ i {lu} c0
  → Process+ i {lu} c1
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)
  → (□fmapRightName : String → String)
  → Process+ i {lu} (c0 ⊕' c1)

```

```

E      (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)          = E L
Lab    (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 x) = Lab L
Lab    (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 x) = Lab L
PE     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 x) = fmapWithLeftName inj1 (PE P x)
PE     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 x) = fmapWithLeftName inj2 (PE P x)
I      (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)          = I P
PI     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 c)
      = PI P c □wNam∞+ Q Using □name , □fmapLeftName , □fmapRightName
PI     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 c)
      = P □wNam+∞ PI Q c Using □name , □fmapLeftName , □fmapRightName
T      (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)          = T L
PT     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 c) = inj1 (PT P c)
PT     (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 c) = inj2 (PT P c)
Str+   (P □wNam+ Q Using □name , □fmapLeftName , □fmapRightName)          = □name

```

```

_□wNam++_Using_,_,_ : {c0 c1 : Choice} → {i : Size}
  → {lu : LUniv}
  → Process+ i {lu} c0
  → Process+ i {lu} c1
  → (□name : String → String → String)
  → (□fmapLeftName : String → String)
  → (□fmapRightName : String → String)
  → Process+ i {lu} (c0 ⊕' c1)

```

```

E      (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName)          = E L
Lab    (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 x) = Lab L
Lab    (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 x) = Lab L
PE     (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 x)
      = fmapWithName∞ □fmapLeftName inj1 (PE P x)
PE     (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 x)

```

```

    = fmapWithName∞ □fmapRightName inj2 (PE Q x)
I    (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName)      = I P ⊕' I Q
PI   (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 c)
    = PI P c □wNam∞++ Q Using □name , □fmapLeftName , □fmapRightName
PI   (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 c)
    = P □wNam+∞+ PI Q c Using □name , □fmapLeftName , □fmapRightName
T    (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName)      = T P ⊕' T Q
PT   (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj1 c) = inj1 (PT P Q)
PT   (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName) (inj2 c) = inj2 (PT P Q)
Str+ (P □wNam++ Q Using □name , □fmapLeftName , □fmapRightName)      = □name (Str P Q)

```

```

_□wNam∞p_Using_,_,_ : {c0 c1 : Choice} → {i : Size}
    → {lu : LUniv}
    → Process∞ i {lu} c0
    → Process i {lu} c1
    → (□name : String → String → String)
    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process∞ i {lu} (c0 ⊕' c1)
forcep (P    □wNam∞p Q Using □name , □fmapLeftName , □fmapRightName)
    = forcep P □wNam Q Using □name , □fmapLeftName , □fmapRightName
Str∞   (P    □wNam∞p Q Using □name , □fmapLeftName , □fmapRightName) = □name (Str P Q)

```

```

_□wNamp∞_Using_,_,_ : {c0 c1 : Choice} → {i : Size}
    → {lu : LUniv}
    → Process i {lu} c0
    → Process∞ i {lu} c1
    → (□name : String → String → String)
    → (□fmapLeftName : String → String)
    → (□fmapRightName : String → String)
    → Process∞ i {lu} (c0 ⊕' c1)
forcep (P    □wNamp∞ Q Using □name , □fmapLeftName , □fmapRightName)
    = P □wNam forcep Q Using □name , □fmapLeftName , □fmapRightName
Str∞   (P    □wNamp∞ Q Using □name , □fmapLeftName , □fmapRightName) = □name (Str P Q)

```

A.37 fdi.agda

```

--@PREFIX@fdi

module fdi where

open import process
open import Size
open import choiceSetU
open import div
open import labelUniv
open import Data.Fin
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import Data.Empty
open import auxData

-- defines the definitions which are identical in fdi and fdiModified

mutual
--@BEGIN@DivergentProcessinf

  record DivergentProcess $\infty$  (i : Size){lu : LUniv}(c : Choice)
    (P : Process $\infty$   $\infty$  {lu} c) : Set where
    coinductive
    field
      forcediv : {j : Size < i}  $\rightarrow$  DivergentProcess j {lu} c (forcep P)

--@END

--@BEGIN@DivergentProcess

  data DivergentProcess (i : Size){lu : LUniv}(c : Choice)
    : (P : Process  $\infty$  {lu} c)  $\rightarrow$  Set where
    div : (P : Process+  $\infty$  c) (divP : DivergentProcess+ i c P)

```

```

    → DivergentProcess i c (node P)

--@END

--@BEGIN@DivergentProcessplus

data DivergentProcess+ (i : Size){lu : LUniv}(c : Choice)
    (P : Process+ ∞ {lu} c) : Set where
  div+ : (int : ChoiceSet (I P))
    (divP : DivergentProcess∞ i c (PI P int))
    → DivergentProcess+ i c P

--@END

open DivergentProcess∞ public

--@BEGIN@exdivergentProcess

divDivergent : {i : Size}{lu : LUniv}(c : Choice) → DivergentProcess∞ i c (DIV∞ {∞} {lu} {c})
forcediv (divDivergent {i} {lu} c) {j} = div DIV+ (div+ zero (divDivergent {j} {lu} c))

--@END

--@BEGIN@traceDivergentinf

data TraceDivergent∞ (i : Size){lu : LUniv}(c : Choice)
    (l : List (Label lu))
    (P : Process∞ ∞ {lu} c) : Set where
  trdiv : (Q : Process ∞ {lu} c) (trp+ : TrP∞ {lu} {c} l (injl Q) P)
    (divp : DivergentProcess i c Q)
    → TraceDivergent∞ i c l P

--@END

--@BEGIN@traceDivergentp

data TraceDivergent (i : Size){lu : LUniv}(c : Choice)
    (l : List (Label lu))
    (P : Process ∞ {lu} c) : Set where

```

```

    trdiv : (Q : Process ∞ {lu} c) (trp : TrP {lu} {c} l (injl Q) P)
            (divp : DivergentProcess i c Q)
            → TraceDivergent i c l P

--@END

--@BEGIN@traceDivergentplus

data TraceDivergent+ (i : Size){lu : LUniv}{c : Choice}
                    (l : List (Label lu)) (P : Process+ ∞ {lu} c) : Set where
    trdiv : (Q : Process ∞ {lu} c)(trp : TrP+ {lu} {c} l (injl Q) P)
            (divp : DivergentProcess i c Q)
            → TraceDivergent+ i c l P

--@END

--@BEGIN@stream

record Stream {i : Size} (X : Set) : Set where
  coinductive
  field
    head : X
    tail : {j : Size < i} → Stream {j} X

--@END

open Stream public

cons : {i : Size}{X : Set}(x : X)(s : Stream {i} X) → Stream {↑ i} X
head (cons x s) = x
tail (cons x s) = s

{- infTr are infinite traces -}

mutual
--@BEGIN@infTraceplus

```

```

data infTr+ {i : Size} {lu : LUniv}{c : Choice}
  : (l : Stream {∞} (Label lu))
    → (P : Process+ ∞ {lu} c) → Set where
extc  : {P : Process+ ∞ {lu} c}
  → (l : Stream {∞} (Label lu))
  → (x : ChoiceSet (E P))
  → (T' (head l == Lab P x))
  → infTr∞ {i} {lu}{c} (tail l) (PE P x)
  → infTr+ {i} {lu}{c} l P
intc  : {P : Process+ ∞ {lu} c}
  → (l : Stream {∞} (Label lu))
  → (x : ChoiceSet (I P))
  → infTr∞' {i} l (PI P x)
  → infTr+ {i} l P

--@END

--@BEGIN@infTracep

data infTr {i : Size} {lu : LUniv}{c : Choice} :
  (l : Stream {∞} (Label lu)) →
  (P : Process ∞ {lu} c) → Set where
tnode : {l : Stream {∞} (Label lu)}
  → {P : Process+ ∞ {lu} c}
  → infTr+ {i} {lu} {c} l P
  → infTr {i} {lu} l (node P)

--@END

--@BEGIN@infTraceinf

record infTr∞ {i : Size} {lu : LUniv}{c : Choice}
  (l : Stream {∞} (Label lu))
  (P : Process∞ ∞ {lu} c) : Set where
  coinductive
  field
    forcetP : {j : Size< i} → infTr {j} l (forcep P)

infTr∞' : {i : Size} {lu : LUniv}{c : Choice}
  (l : Stream {∞} (Label lu))

```



```

      (P : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  Set
    infTr $\infty$ ' {i} {lu} {c} l P = infTr {i} l (forcep P)

--@END

open infTr $\infty$  public

--@BEGIN@exampleInfprocess

-- a  $\rightarrow$  a  $\rightarrow$  a ....

mutual

  inflP $\infty$  : {i : Size}  $\rightarrow$  {lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  (l : (Label lu))  $\rightarrow$  Process $\infty$  i c
  forcep (inflP $\infty$  l) = inflP l
  Str $\infty$  (inflP $\infty$  l) = "inflP"

  inflP : {i : Size}  $\rightarrow$  {lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  (l : (Label lu))  $\rightarrow$  Process i c
  inflP l = node (inflP+ l)

  inflP+ : {i : Size}  $\rightarrow$  {lu : LUniv}  $\rightarrow$  {c : Choice}  $\rightarrow$  (l : (Label lu))  $\rightarrow$  Process+ i c
  E (inflP+ {i} {c} l) = fin 1
  Lab (inflP+ {i} {c} l) _ = l
  PE (inflP+ {i} {c} l) _ = inflP $\infty$  l
  I (inflP+ {i} {c} l) =  $\emptyset$ '
  PI (inflP+ {i} {c} l) ()
  T (inflP+ {i} {c} l) =  $\emptyset$ '
  PT (inflP+ {i} {c} l) ()
  Str+ (inflP+ {i} {c} l) = "inflP+ 1"

  infl : {lu : LUniv} (l : (Label lu))  $\rightarrow$  Stream { $\infty$ } (Label lu)
  head (infl l) = l
  tail (infl l) = infl l

--@END

--@BEGIN@exampleInfTrace

```



mutual

```
infTraceInflP $\infty$  : {lu : LUniv}{c : Choice}(l : (Label lu))  $\rightarrow$  infTr $\infty$  { $\infty$ } {lu} {c} (infl l) (inflP $\infty$  l)
forcetP (infTraceInflP $\infty$  {lu} {c} l) {j} = infTraceInflP {lu} {c} l
```

```
infTraceInflP : {lu : LUniv}{c : Choice}(l : (Label lu))  $\rightarrow$  infTr { $\infty$ } {lu} {c} (infl l) (inflP l)
infTraceInflP {lu} {c} l = tnode (infTraceInflP+ l)
```

```
infTraceInflP+ : {lu : LUniv}{c : Choice}(l : (Label lu))  $\rightarrow$  infTr+ { $\infty$ } {lu} {c} (infl l) (inflP+ l)
infTraceInflP+ {lu} {c} l = extc (infl l) zero (refl==l {lu} {l = l}) (infTraceInflP $\infty$  {lu} {c} l)
```

--@END

mutual

--@BEGIN@processStableInfSch

```
stableSch $\infty$  : {lu : LUniv}{c : Choice}(P : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  Set
stableSch $\infty$  P = stableSch (forcetP P)
```

--@END

--@BEGIN@processStableSch

```
stableSch+ : {lu : LUniv}{c : Choice}(P : Process+  $\infty$  {lu} c)  $\rightarrow$  Set
stableSch+ P =  $\neg$  (ChoiceSet (I P))
```

```
stableSch : {lu : LUniv}{c : Choice}(P : Process  $\infty$  {lu} c)  $\rightarrow$  Set
stableSch (terminate x) =  $\top$ 
stableSch (node P) = stableSch+ P
```

--@END

--In this def we follow Schneider book Page 172
 -- that a process is stable if it has no tau transitions
 -- where tick events are no tau transitions

-- the above is Schneider stability.
 -- Roscoe stability wouldmean
 --

-- stable+ : {c : Choice}(P : Process+ ∞ {lu} c) \rightarrow Set

```

-- stable+ P = ((int : ChoiceSet (I P)) → ⊥ ) ×
-- ((t : ChoiceSet (T P)) → ⊥ )
--
-- stable : {c : Choice}(P : Process ∞ {lu} c) → Set
-- stable {c} (terminate x) = ⊥
-- stable {c} (node P) = stable+ P

--@BEGIN@NoTickIfRoscoe

noTickIfRoscoe+ : {lu : LUniv}{c : Choice}(isRoscoe : Bool)
                  (P : Process+ ∞ {lu} c) → Set
noTickIfRoscoe+ false P = ⊤
noTickIfRoscoe+ true P = ¬ (ChoiceSet (T P))

noTickIfRoscoe : {lu : LUniv}{c : Choice}(isRoscoe : Bool)
                  (P : Process ∞ {lu} c)
                  → Set
noTickIfRoscoe false (terminate x) = ⊤
noTickIfRoscoe true (terminate x) = ⊥
noTickIfRoscoe isRoscoe (node Q) = noTickIfRoscoe+ isRoscoe Q

--@END

mutual
--@BEGIN@processStableInf

stableParametrized∞ : {lu : LUniv}{c : Choice}(isRoscoe : Bool)
                     (P : Process∞ ∞ {lu} c) → Set
stableParametrized∞ b P = stableParametrized b (forcep P)

--@END

--@BEGIN@processStable

stableParametrized+ : {lu : LUniv}{c : Choice}(isRoscoe : Bool)
                     (P : Process+ ∞ {lu} c) → Set
stableParametrized+ isRoscoe P = stableSch+ P ×
                               noTickIfRoscoe+ isRoscoe P

stableParametrized : {lu : LUniv}{c : Choice}(isRoscoe : Bool)

```

```

      (P : Process ∞ {lu} c) → Set
    stableParametrized {c} isRoscoe (terminate x) = ¬ (T' isRoscoe)
    stableParametrized {c} b (node P) = stableParametrized+ b P

--@END

--@BEGIN@processStableUnparametrized

stable∞ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c) → Set
stable∞ P = stableParametrized∞ true P

stable+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c) → Set
stable+ P = stableParametrized+ true P

stable : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c) → Set
stable P = stableParametrized true P

--@END

--@BEGIN@stabToNoInternalChoice

stabToNoInternal+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (stab : stable+ P)
  → ¬ (ChoiceSet (I P))
stabToNoInternal+ P (noInterCh „ notermEv) intChoice = noInterCh intChoice

--@END

stabSchNoTickIfRos2StablePar+ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (isRoscoe : Bool)
  (stab : stableSch+ P)
  (notick : noTickIfRoscoe+ isRoscoe P)
  → stableParametrized+ isRoscoe P
stabSchNoTickIfRos2StablePar+ P isRoscoe stab notick = stab „ notick

--@BEGIN@stabSchNoTickIfRosTwoStablePar

stabSchNoTickIfRos2StablePar : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)

```

```

      (isRoscoe : Bool)
      (stabSch : stableSch P)
      (notick : noTickIfRoscoe isRoscoe P)
      → stableParametrized isRoscoe P
stabSchNoTickIfRos2StablePar (terminate x) false stabSch notick ()
stabSchNoTickIfRos2StablePar (terminate x) true stabSch ()
stabSchNoTickIfRos2StablePar (node x) false stabSch notick = stabSch „ –
stabSchNoTickIfRos2StablePar (node x) true stabSch notick = stabSch „ notick

--@END

```

A.38 fdiImpliesEquivalence.agda

```

module fdiImpliesEquivalence where

open import process
open import choiceSetU
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Sum
open import renamingResult
open import lemFmap
open import auxData
open import RefWithoutSize
open import bisimilarity
open import bisimSImpliesBisimw
open import bisimilarityProofs
open import bisimImpliesTraceEquiv
open import bisimLemFmap
open import bisimImpliesFDI
open import fdiRefusal
open import bisimImpliesBisim
open import externalChoice
open import addTick
open import labelUniv

```

A.39 fdiOld.agda

```
--@PREFIX@fdi

-- files which were in fdi.agda before renaming fdiCommonNormalAndModified.agda
-- to fdi

module fdiOld where

open import process
open import Size
open import choiceSetU
open import div
open import labelUniv
open import Data.Fin
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import Data.Empty
open import auxData
```

A.40 fdiPart2.agda

```
--@PREFIX@mainfdiPartTwo

module fdiPart2 where

open import process
open import Size
open import choiceSetU
open import primitiveProcess
open import div
open import Data.Fin
open import Data.List
open import Data.Sum
```

```

open import TraceWithNextProcess
open import dataAuxFunction
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import Data.Maybe
open import dataAuxFunction
open import Data.Empty
open import TraceWithoutSize
open import RefWithoutSize
open import auxData
open import labelUniv
open import fdi

--@BEGIN@refusalsinf

data refusal∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP∞ {lu} {c} [] (inj1 Q) P)
    (stab : stable Q)
    (Xreject : (l : (Label lu)) → (T'(X l))
      → ¬ (Tr (l :: []) nothing Q))
    → refusal∞ P X

--@END

--@BEGIN@refusalsp

data refusal {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP {lu} {c} [] (inj1 Q) P)
    (stab : stable Q)
    (Xreject : (l : (Label lu)) → (T'(X l))
      → ¬ (Tr (l :: []) nothing Q))
    → refusal P X

```

```
--@END
```

```
--@BEGIN@refusalsPlus
```

```
data refusal+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
    (tr : TrP+ {lu} {c} [] (injl Q) P)
    (stab : stable Q)
    (Xreject : (l : (Label lu)) → (T'(X l))
      → ¬ (Tr (l :: []) nothing Q))
    → refusal+ P X
```

```
--@END
```

```
--@BEGIN@stableFailureinf
```

```
data stableFailure∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFp : (Q : Process ∞ {lu} c)
    (tr : TrP∞ {lu} {c} l (injl Q) P)
    (stab : stable Q)
    (refuse : refusal Q X)
    → stableFailure∞ P l X
```

```
--@END
```

```
--@BEGIN@stableFailurep
```

```
data stableFailure {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFp : (Q : Process ∞ {lu} c)
    (tr : TrP {lu} {c} l (injl Q) P)
    (stab : stable Q)
    (refuse : refusal Q X)
    → stableFailure P l X
```

```
--@END
```

```
--@BEGIN@stableFailurePlus
```

```
data stableFailure+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFp : (Q : Process ∞ {lu} c)
    (tr : TrP+ {lu} {c} l (injl Q) P)
    (stab : stable Q)
    (refuse : refusal Q X)
    → stableFailure+ P l X
```

```
--@END
```

```
--@BEGIN@failureinf
```

```
data failure∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFail : stableFailure∞ P l X
    → failure∞ P l X
  divergentFailure : TraceDivergent∞ ∞ c l P
    → failure∞ P l X
```

```
--@END
```

```
--@BEGIN@failurePlus
```

```
data failure+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFail : stableFailure+ P l X
    → failure+ P l X
  divergentFailure : TraceDivergent+ ∞ c l P
    → failure+ P l X
```

```
--@END
```

```
--@BEGIN@failurep
```

```
data failure {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))
  (X : (Label lu) → Bool) : Set where
  stableFail : stableFailure P l X
                → failure P l X
  divergentFailure : TraceDivergent ∞ c l P
                    → failure P l X
```

```
--@END
```

```
--@BEGIN@SFRP
```

```
_⊑sf₁_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
        (Q : Process ∞ {lu} c) → Set
_⊑sf₁_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
                    → stableFailure Q l X
                    → stableFailure P l X
```

```
--@END
```

```
--@BEGIN@SFRPlus
```

```
_⊑sf₁+_ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
        (Q : Process+ ∞ {lu} c) → Set
_⊑sf₁+_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
                    → stableFailure+ Q l X
                    → stableFailure+ P l X
```

```
--@END
```

```
--@BEGIN@SFRinf
```

```
_⊑sf₁∞_ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
        (Q : Process∞ ∞ {lu} c) → Set
_⊑sf₁∞_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
```

```

→ stableFailure∞ Q l X
→ stableFailure∞ P l X

--@END

--@BEGIN@SF

_⊑sf_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
      (Q : Process ∞ {lu} c) → Set
P ⊑sf Q = (P ⊑ Q) × (P ⊑sf1 Q)

--@END

--@BEGIN@fdi

_⊑fdi1_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
          (Q : Process ∞ {lu} c) → Set
_⊑fdi1_ {lu} {c} P Q = (l : List (Label lu)) → TraceDivergent ∞ c l Q
                    → TraceDivergent ∞ c l P

--@END

--@BEGIN@fdit

_⊑fdi2_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
          (Q : Process ∞ {lu} c) → Set
_⊑fdi2_ {lu} {c} P Q = (l : List (Label lu))(X : (Label lu) → Bool)
                    → stableFailure Q l X → stableFailure P l X

_⊑fdi2+_ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
          (Q : Process+ ∞ {lu} c) → Set
_⊑fdi2+_ {lu} {c} P Q = (l : List (Label lu))(X : (Label lu) → Bool)
                    → stableFailure+ Q l X → stableFailure+ P l X

_⊑fdi2∞_ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
          (Q : Process∞ ∞ {lu} c) → Set
_⊑fdi2∞_ {lu} {c} P Q = (l : List (Label lu))(X : (Label lu) → Bool)
                    → stableFailure∞ Q l X → stableFailure∞ P l X

--@END

```

```

--@BEGIN@fdiref

_⊑fdi_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
      (Q : Process ∞ {lu} c) → Set
P ⊑fdi Q = ((P ⊑ Q) × (P ⊑fdi1 Q)) × (P ⊑fdi2 Q)

_≡fdi_ : {lu : LUniv}{c0 : Choice} → (P Q : Process ∞ {lu} c0) → Set
P ≡fdi Q = (P ⊑fdi Q) × (Q ⊑fdi P)

--@END

```

A.41 fdiRefusal.agda

```

--@PREFIX@mainfdiRefusal

module fdiRefusal where

open import process
open import Size
open import choiceSetU
open import primitiveProcess
open import div
open import labelUniv
open import Data.Fin
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import Data.Maybe
open import dataAuxFunction
open import Data.Empty
open import TraceWithoutSize
open import RefWithoutSize
open import auxData

```

```

open import fdi

--In this def we follow Schneider book Page 172
-- that a process is stable if it has no tau transitions
-- where tick events are no tau transitions

-- if we had Q : Process+
-- then we could write
-- Xreject : (e : E Q) -> ¬ (T' (X ((Label lu) Q e)))
--
-- however Q : Process
-- but you can define

--- Ep : Process -> ChoiceSet
-- Ep (terminate a) = emptyset
-- Ep (node Q) = E Q

-- same for (Label lu)

--- labelp : (Q : Process) -> ChoiceSet (Ep Q) -> (Label lu)
-- labelp (terminate a) ()
-- labelp (node Q) a = label Q a

-- dRefusal for direct refusal

--@BEGIN@NoExtChInX

NoExtChInX : {lu : LUniv}{c : Choice}(Q : Process+ ∞ c)
            (X : (Label lu) → Bool)
            → Set
NoExtChInX Q X = (e : ChoiceSet (E Q)) → ¬ (T'(X (Lab Q e)))

--@END

--@BEGIN@NoTicksIfIsRoscoe

```

```

NoTicksIfIsRoscoe : {lu : LUniv}{c : Choice}(Q : Process+ ∞ {lu} c)
  (isRoscoe : Bool)
  → Set
NoTicksIfIsRoscoe Q isRoscoe = (tickIsIncl : T' isRoscoe)
  → ¬ (ChoiceSet (T Q))

--@END

--@BEGIN@DRefusal

data DRefusal+ {lu : LUniv}{c : Choice}(Q : Process+ ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  drefusal : (noextChInX : NoExtChInX Q X)
    (noTerm : NoTicksIfIsRoscoe Q isRoscoe)
    → DRefusal+ {lu}{c} Q isRoscoe X

DRefusal : {lu : LUniv}{c : Choice}(Q : Process ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) → Set
DRefusal {lu}{c} (terminate x) isRoscoe X = ¬ (T' isRoscoe)
DRefusal {lu}{c} (node x) isRoscoe X = DRefusal+ {lu}{c} x isRoscoe X

DRefusal∞ : {lu : LUniv}{c : Choice}(Q : Process∞ ∞ {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) → Set
DRefusal∞ {lu}{c} Q isRoscoe X = DRefusal {lu}{c} (forcep Q) isRoscoe X

--@END

-- old version:
--
-- (Xreject : (l : (Label lu)) → (T'(X l)))
-- → ¬ (Tr (l :: []) nothing Q))
-- (terreject : (T' isRoscoe)
-- → (x : ChoiceSet c)
-- → ¬ (Tr [] (just x) Q))
-- if Q is stable the two definitions should be equivalent

```

```

-- -- Old versions of refusal with commented out code which is now left out
--
-- data refusal {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)(isRoscoe :
-- (X : (Label lu) → Bool) : Set where
-- refusalp : (Q : Process ∞ {lu} c)
-- (tr : TrP {lu}{c} [] (inj1 Q) P)
-- (stab : stable Q)
-- (drefuse : DRefusal Q isRoscoe X)
-- {- (Xreject : (l : (Label lu)) → (T'(X l))
-- → ¬ (Tr (l :: []) nothing Q))
-- (terreject : (T' isRoscoe)
-- → (x : ChoiceSet c)
-- → ¬ (Tr [] (just x) Q))-}
-- → refusal P isRoscoe X
--
--
-- data refusal+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(isRoscoe
-- (X : (Label lu) → Bool) : Set where
-- refusalp : (Q : Process ∞ {lu} c)
-- (tr : TrP+ {lu}{c} [] (inj1 Q) P)
-- (stab : stable Q)
-- (drefuse : DRefusal Q isRoscoe X)
-- {- (Xreject : (l : (Label lu)) → (T'(X l))
-- → ¬ (Tr (l :: []) nothing Q))
-- (terreject : (T' isRoscoe)
-- → (x : ChoiceSet c)
-- → ¬ (Tr [] (just x) Q))-}
-- → refusal+ P isRoscoe X
--
--
-- data refusal∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)(isRoscoe
-- (X : (Label lu) → Bool) : Set where
-- refusalp : (Q : Process ∞ {lu} c)
-- (tr : TrP∞ {lu}{c} [] (inj1 Q) P)
-- (stab : stable Q)
-- (drefuse : DRefusal Q isRoscoe X)

```

```

-- {- (Xreject : (l : (Label lu)) → (T' (X l)))
-- → ¬ (Tr (l :: []) nothing Q))
-- (terreject : (T' isRoscoe)
-- → (x : ChoiceSet c)
-- → ¬ (Tr [] (just x) Q))
-- -}
-- → refusal∞ P isRoscoe X
--
--

--@BEGIN@refusalsp

data refusal    {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)(isRoscoe : Bool)
                (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
              (tr : TrP {lu}{c} [] (injl Q) P)
              (stab : stable Q)
              (drefuse : DRefusal Q isRoscoe X)
              → refusal P isRoscoe X

--@END

--@BEGIN@refusalsPlus

data refusal+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(isRoscoe : Bool)
              (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)
              (tr : TrP+ {lu}{c} [] (injl Q) P)
              (stab : stable Q)
              (drefuse : DRefusal Q isRoscoe X)
              → refusal+ P isRoscoe X

--@END

--@BEGIN@refusalsinf

data refusal∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)(isRoscoe : Bool)
              (X : (Label lu) → Bool) : Set where
  refusalp : (Q : Process ∞ {lu} c)

```

```

      (tr : TrP $\infty$  {lu}{c} [] (inj1 Q) P)
      (stab : stable Q)
      (drefuse : DRefusal Q isRoscoe X)
      → refusal $\infty$  P isRoscoe X

--@END

--@BEGIN@stableFailureinf

data stableFailure $\infty$  {lu : LUniv}{c : Choice}(P : Process $\infty$  {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFp : (Q : Process $\infty$  {lu} c)
    (tr : TrP $\infty$  {lu}{c} l (inj1 Q) P)
    (stab : stable Q)
    (drefuse : DRefusal Q isRoscoe X)
    → stableFailure $\infty$  P l isRoscoe X

--@END

--@BEGIN@stableFailurep

data stableFailure {lu : LUniv}{c : Choice}(P : Process $\infty$  {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFp : (Q : Process $\infty$  {lu} c)
    (tr : TrP {lu}{c} l (inj1 Q) P)
    (stab : stable Q)
    (drefuse : DRefusal Q isRoscoe X)
    → stableFailure P l isRoscoe X

--@END

--@BEGIN@stableFailurePlus

data stableFailure+ {lu : LUniv}{c : Choice}(P : Process+ $\infty$  {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)

```

```

      (X : (Label lu) → Bool) : Set where
    stableFp : (Q : Process ∞ {lu} c)
      (tr : TrP+ {lu}{c} l (injl Q) P)
      (stab : stable Q)
      (drefuse : DRefusal Q isRoscoe X)
      → stableFailure+ P l isRoscoe X

--@END

--@BEGIN@failureinf

data failure∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
    stableFail : stableFailure∞ P l isRoscoe X
      → failure∞ P l isRoscoe X
    divergentFailure : TraceDivergent∞ ∞ c l P
      → failure∞ P l isRoscoe X

--@END

--@BEGIN@failurePlus

data failure+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
    stableFail : stableFailure+ P l isRoscoe X
      → failure+ P l isRoscoe X
    divergentFailure : TraceDivergent+ ∞ c l P
      → failure+ P l isRoscoe X

--@END

--@BEGIN@failurep

data failure {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))

```

```

      (isRoscoe : Bool)
      (X : (Label lu) → Bool) : Set where
stableFail : stableFailure P l isRoscoe X
             → failure P l isRoscoe X
divergentFailure : TraceDivergent ∞ c l P
                  → failure P l isRoscoe X

--@END

--@BEGIN@SFRP

_⊑sf₁_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
        (Q : Process ∞ {lu} c) → Set
_⊑sf₁_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
                    → stableFailure Q l true X
                    → stableFailure P l true X

--@END

--@BEGIN@SFRPlus

_⊑sf₁+_ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
          (Q : Process+ ∞ {lu} c) → Set
_⊑sf₁+_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
                    → stableFailure+ Q l true X
                    → stableFailure+ P l true X

--@END

--@BEGIN@SFRinf

_⊑sf₁∞_ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
          (Q : Process∞ ∞ {lu} c) → Set
_⊑sf₁∞_ {lu}{c} P Q = (l : List (Label lu)) (X : (Label lu) → Bool)
                    → stableFailure∞ Q l true X
                    → stableFailure∞ P l true X

--@END

--@BEGIN@SF

```

```

 $\_ \sqsubseteq \text{sf} \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$ 
 $(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P \sqsubseteq \text{sf} Q = (P \sqsubseteq Q) \times (P \sqsubseteq \text{sf}_1 Q)$ 

```

```

 $\_ = \text{sf} \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$ 
 $(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P = \text{sf} Q = (P \sqsubseteq \text{sf} Q) \times (Q \sqsubseteq \text{sf} P)$ 

```

```
--@END
```

```

 $\_ \sqsubseteq \text{sf} + \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} + \infty \{lu\} c)$ 
 $(Q : \text{Process} + \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P \sqsubseteq \text{sf} + Q = (P \sqsubseteq + Q) \times (P \sqsubseteq \text{sf}_1 + Q)$ 

```

```

 $\_ \sqsubseteq \text{sf} \infty \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \infty \{lu\} c)$ 
 $(Q : \text{Process} \infty \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P \sqsubseteq \text{sf} \infty Q = (P \sqsubseteq \infty Q) \times (P \sqsubseteq \text{sf}_1 \infty Q)$ 

```

```

 $\_ = \text{sf} + \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} + \infty \{lu\} c)$ 
 $(Q : \text{Process} + \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P = \text{sf} + Q = (P \sqsubseteq \text{sf} + Q) \times (Q \sqsubseteq \text{sf} + P)$ 

```

```

 $\_ = \text{sf} \infty \_ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \infty \{lu\} c)$ 
 $(Q : \text{Process} \infty \infty \{lu\} c) \rightarrow \text{Set}$ 
 $P = \text{sf} \infty Q = (P \sqsubseteq \text{sf} \infty Q) \times (Q \sqsubseteq \text{sf} \infty P)$ 

```

```

-- data TraceDivergentNoTr $\infty$  (i : Size){lu : LUniv}(c : Choice)
-- (P : Process $\infty$   $\infty$  {lu} c) : Set where
-- trdiv : (Q : Process  $\infty$  {lu} c)
-- (trp+ : TrP $\infty$  {lu}{c} l (inj1 Q) P)
-- (divp : DivergentProcess i c Q)
--  $\rightarrow$  TraceDivergentNoTr $\infty$  i c l P

```

```

-- data TraceDivergentNoTr (i : Size){lu : LUniv}(c : Choice)(l : List (Label lu))
-- (P : Process  $\infty$  {lu} c) : Set where

```

```

-- trdiv : (Q : Process  $\infty$  {lu} c)
-- (trp : TrP {lu}{c} l (injl Q) P)
-- (divp : DivergentProcess i c Q)
--  $\rightarrow$  TraceDivergentNoTr i c l P

-- data TraceDivergentNoTr+ (i : Size){lu : LUniv}(c : Choice)(l : List (
-- (P : Process+  $\infty$  {lu} c) : Set where
-- trdiv : (Q : Process  $\infty$  {lu} c)
-- (trp : TrP+ {lu}{c} l (injl Q) P)
-- (divp : DivergentProcess i c Q)
--  $\rightarrow$  TraceDivergentNoTr+ i c l P

```

```

data stableFailureNoTr $\infty$  {lu : LUniv}{c : Choice}(P : Process $\infty$   $\infty$  {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu)  $\rightarrow$  Bool) : Set where
  stableFp : (stab : stable $\infty$  P)
    (refuse : refusal $\infty$  P isRoscoe X)
     $\rightarrow$  stableFailureNoTr $\infty$  P isRoscoe X

```

```

data stableFailureNoTr {lu : LUniv}{c : Choice}(P : Process  $\infty$  {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu)  $\rightarrow$  Bool) : Set where
  stableFp : (stab : stable P)
    (refuse : refusal P isRoscoe X)
     $\rightarrow$  stableFailureNoTr P isRoscoe X

```

```

data stableFailureNoTr+ {lu : LUniv}{c : Choice}(P : Process+  $\infty$  {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu)  $\rightarrow$  Bool) : Set where
  stableFp : (stab : stable+ P)
    (refuse : refusal+ P isRoscoe X)
     $\rightarrow$  stableFailureNoTr+ P isRoscoe X

```

```

data failureNoTr $\infty$  {lu : LUniv}{c : Choice}(P : Process $\infty$   $\infty$  {lu} c)
  (isRoscoe : Bool)

```

```

      (X : (Label lu) → Bool) : Set where
stableFail : stableFailureNoTr $\infty$  P isRoscoe X
              → failureNoTr $\infty$  P isRoscoe X
divergentFailure : DivergentProcess $\infty$   $\infty$  {lu} c P
                  → failureNoTr $\infty$  P isRoscoe X

data failureNoTr+ {lu : LUniv}{c : Choice}(P : Process+  $\infty$  {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
stableFail : stableFailureNoTr+ P isRoscoe X
              → failureNoTr+ P isRoscoe X
divergentFailure : DivergentProcess+  $\infty$  {lu} c P
                  → failureNoTr+ P isRoscoe X

data failureNoTr {lu : LUniv}{c : Choice}(P : Process  $\infty$  {lu} c)
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
stableFail : stableFailureNoTr P isRoscoe X
              → failureNoTr P isRoscoe X
divergentFailure : DivergentProcess  $\infty$  {lu} c P
                  → failureNoTr P isRoscoe X

data failureWithTr $\infty$  {lu : LUniv}{c : Choice}(P : Process $\infty$   $\infty$  {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
failWithTr : (Q : Process  $\infty$  {lu} c)
              (fail : failureNoTr Q isRoscoe X)
              (tr : TrP $\infty$  l (inj1 Q) P)
              → failureWithTr $\infty$  P l isRoscoe X

data failureWithTr {lu : LUniv}{c : Choice}(P : Process  $\infty$  {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
failWithTr : (Q : Process  $\infty$  {lu} c)
              (fail : failureNoTr Q isRoscoe X)

```

$$(tr : \text{TrP } l (\text{inj}_1 Q) P) \\ \rightarrow \text{failureWithTr } P l \text{ isRoscoe } X$$

```
data failureWithTr+      {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  failWithTr : (Q : Process ∞ {lu} c)
    (fail : failureNoTr Q isRoscoe X)
    (tr : TrP+ l (inj1 Q) P)
    → failureWithTr+ P l isRoscoe X
```

```
data stableFailureWithTr∞ {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFailWithTr : (Q : Process ∞ {lu} c)
    (fail : stableFailureNoTr Q isRoscoe X)
    (tr : TrP∞ l (inj1 Q) P)
    → stableFailureWithTr∞ P l isRoscoe X
```

```
data stableFailureWithTr {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFailWithTr : (Q : Process ∞ {lu} c)
    (fail : stableFailureNoTr Q isRoscoe X)
    (tr : TrP l (inj1 Q) P)
    → stableFailureWithTr P l isRoscoe X
```

```
data stableFailureWithTr+ {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)
  (l : List (Label lu))
  (isRoscoe : Bool)
  (X : (Label lu) → Bool) : Set where
  stableFailWithTr : (Q : Process ∞ {lu} c)
    (fail : stableFailureNoTr Q isRoscoe X)
    (tr : TrP+ l (inj1 Q) P)
```

→ `stableFailureWithTr+` $P\ l\ isRoscoe\ X$

--@BEGIN@fdi

`_⊑fdi_` : $\{lu : LUniv\}\{c : Choice\} (P : Process \infty \{lu\}\ c)$
 $(Q : Process \infty \{lu\}\ c) \rightarrow Set$
`_⊑fdi_` $\{lu\}\{c\}\ P\ Q = (l : List\ (Label\ lu))$
 → `TraceDivergent` $\infty\ c\ l\ Q$
 → `TraceDivergent` $\infty\ c\ l\ P$

--@END

`_⊑fdi+_` : $\{lu : LUniv\}\{c : Choice\} (P : Process+ \infty \{lu\}\ c)$
 $(Q : Process+ \infty \{lu\}\ c) \rightarrow Set$
`_⊑fdi+_` $\{lu\}\{c\}\ P\ Q = (l : List\ (Label\ lu))$
 → `TraceDivergent+` $\infty\ c\ l\ Q$
 → `TraceDivergent+` $\infty\ c\ l\ P$

`_⊑fdi∞_` : $\{lu : LUniv\}\{c : Choice\} (P : Process\infty \infty \{lu\}\ c)$
 $(Q : Process\infty \infty \{lu\}\ c) \rightarrow Set$
`_⊑fdi∞_` $\{lu\}\{c\}\ P\ Q = (l : List\ (Label\ lu))$
 → `TraceDivergent∞` $\infty\ c\ l\ Q$
 → `TraceDivergent∞` $\infty\ c\ l\ P$

--@BEGIN@fdit

`_⊑fdi2ros_` : $\{lu : LUniv\}\{c : Choice\} (P : Process \infty \{lu\}\ c)$
 $(Q : Process \infty \{lu\}\ c) \rightarrow Set$
`_⊑fdi2ros_` $\{lu\}\{c\}\ P\ Q = (l : List\ (Label\ lu))$
 $(X : (Label\ lu) \rightarrow Bool)$
 → `failure` $Q\ l\ true\ X$
 → `failure` $P\ l\ true\ X$

--@END

`_⊑fdi2ros+_` : $\{lu : LUniv\}\{c : Choice\} (P : Process+ \infty \{lu\}\ c)$
 $(Q : Process+ \infty \{lu\}\ c) \rightarrow Set$
`_⊑fdi2ros+_` $\{lu\}\{c\}\ P\ Q = (l : List\ (Label\ lu))$

```

(X : (Label lu) → Bool)
  → failure+ Q l true X
  → failure+ P l true X

```

```

_⊑fdi2ros∞_ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
              (Q : Process∞ ∞ {lu} c) → Set
_⊑fdi2ros∞_ {lu}{c} P Q = (l : List (Label lu))
              (X : (Label lu) → Bool)
              → failure∞ Q l true X
              → failure∞ P l true X

```

```
--@BEGIN@fdiInfTraces
```

```

_⊑fdi3_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)
          (Q : Process ∞ {lu} c) → Set
_⊑fdi3_ {lu}{c} P Q = (l : Stream {∞} (Label lu))
          (tr : infTr {∞} l Q)
          → infTr {∞} l P

```

```
--@END
```

```

_⊑fdi3+_ : {lu : LUniv}{c : Choice} (P : Process+ ∞ {lu} c)
          (Q : Process+ ∞ {lu} c) → Set
_⊑fdi3+_ {lu}{c} P Q = (l : Stream {∞} (Label lu))
          (tr : infTr+ {∞} l Q)
          → infTr+ {∞} l P

```

```

_⊑fdi3∞_ : {lu : LUniv}{c : Choice} (P : Process∞ ∞ {lu} c)
          (Q : Process∞ ∞ {lu} c) → Set
_⊑fdi3∞_ {lu}{c} P Q = (l : Stream {∞} (Label lu))
          (tr : infTr∞ {∞} l Q)
          → infTr∞ {∞} l P

```

```
--@BEGIN@fdiref
```

```

_⊑fdi_ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c)

```

$$(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$$

$$P \sqsubseteq_{\text{fdi}} Q = (((P \sqsubseteq Q) \times (P \sqsubseteq_{\text{fdi}_1} Q)) \times (P \sqsubseteq_{\text{fdi}_2\text{ros}} Q)) \times (P \sqsubseteq_{\text{fdi}_3} Q)$$

$$_ \equiv_{\text{fdi}} _ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P Q : \text{Process} \infty \{lu\} c_0) \rightarrow \text{Set}$$

$$P \equiv_{\text{fdi}} Q = (P \sqsubseteq_{\text{fdi}} Q) \times (Q \sqsubseteq_{\text{fdi}} P)$$

--@END

$$_ \sqsubseteq_{\text{fdi}+} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c)$$

$$(Q : \text{Process}+ \infty \{lu\} c) \rightarrow \text{Set}$$

$$P \sqsubseteq_{\text{fdi}+} Q = (((P \sqsubseteq+ Q) \times (P \sqsubseteq_{\text{fdi}_1+} Q)) \times (P \sqsubseteq_{\text{fdi}_2\text{ros}+} Q)) \times (P \sqsubseteq_{\text{fdi}_3+} Q)$$

$$_ \sqsubseteq_{\text{fdi}\infty} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c)$$

$$(Q : \text{Process}\infty \infty \{lu\} c) \rightarrow \text{Set}$$

$$P \sqsubseteq_{\text{fdi}\infty} Q = (((P \sqsubseteq\infty Q) \times (P \sqsubseteq_{\text{fdi}_1\infty} Q)) \times (P \sqsubseteq_{\text{fdi}_2\text{ros}\infty} Q)) \times (P \sqsubseteq_{\text{fdi}_3\infty} Q)$$

$$_ \equiv_{\text{fdi}+} _ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P Q : \text{Process}+ \infty \{lu\} c_0) \rightarrow \text{Set}$$

$$P \equiv_{\text{fdi}+} Q = (P \sqsubseteq_{\text{fdi}+} Q) \times (Q \sqsubseteq_{\text{fdi}+} P)$$

$$_ \equiv_{\text{fdi}\infty} _ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P Q : \text{Process}\infty \infty \{lu\} c_0) \rightarrow \text{Set}$$

$$P \equiv_{\text{fdi}\infty} Q = (P \sqsubseteq_{\text{fdi}\infty} Q) \times (Q \sqsubseteq_{\text{fdi}\infty} P)$$

$$_ \sqsubseteq_{\text{fdi}_2\text{sch}} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$$

$$(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$$

$$_ \sqsubseteq_{\text{fdi}_2\text{sch}} _ \{lu\} \{c\} P Q = (l : \text{List} (\text{Label } lu))$$

$$(X : (\text{Label } lu) \rightarrow \text{Bool})$$

$$\rightarrow \text{failure } Q \text{ l false } X$$

$$\rightarrow \text{failure } P \text{ l false } X$$

$$_ \sqsubseteq_{\text{fdisch}} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$$

$$(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$$

$$P \sqsubseteq_{\text{fdisch}} Q = ((P \sqsubseteq Q) \times (P \sqsubseteq_{\text{fdi}_1} Q)) \times (P \sqsubseteq_{\text{fdi}_2\text{sch}} Q)$$

$$_ \sqsubseteq_{\text{fdi}_2\text{rosaux}} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$$

$$(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$$

$$_ \sqsubseteq_{\text{fdi}_2\text{rosaux}} _ \{lu\} \{c\} P Q = (l : \text{List} (\text{Label } lu))$$

$$(X : (\text{Label } lu) \rightarrow \text{Bool})$$

$\rightarrow \text{failure } Q \text{ l true } X$
 $\rightarrow \text{failure } P \text{ l true } X$

$_ \sqsubseteq \text{fdiold} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$
 $(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$
 $P \sqsubseteq \text{fdiold } Q = ((P \sqsubseteq Q) \times (P \sqsubseteq \text{fdi}_1 Q)) \times (P \sqsubseteq \text{fdi}_2 \text{ros } Q)$

$_ \sqsubseteq \text{fdi}\infty\text{old} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c)$
 $(Q : \text{Process}\infty \infty \{lu\} c) \rightarrow \text{Set}$
 $P \sqsubseteq \text{fdi}\infty\text{old } Q = ((P \sqsubseteq \infty Q) \times (P \sqsubseteq \text{fdi}_1 \infty Q)) \times (P \sqsubseteq \text{fdi}_2 \text{ros}\infty Q)$

$_ \sqsubseteq \text{fdi+old} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+} \infty \{lu\} c)$
 $(Q : \text{Process+} \infty \{lu\} c) \rightarrow \text{Set}$
 $P \sqsubseteq \text{fdi+old } Q = ((P \sqsubseteq + Q) \times (P \sqsubseteq \text{fdi}_1 + Q)) \times (P \sqsubseteq \text{fdi}_2 \text{ros+ } Q)$

$_ = \text{fdiold} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$
 $(Q : \text{Process} \infty \{lu\} c) \rightarrow \text{Set}$
 $P = \text{fdiold } Q = (P \sqsubseteq \text{fdiold } Q) \times (Q \sqsubseteq \text{fdiold } P)$

$_ = \text{fdi+old} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process+} \infty \{lu\} c)$
 $(Q : \text{Process+} \infty \{lu\} c) \rightarrow \text{Set}$
 $P = \text{fdi+old } Q = (P \sqsubseteq \text{fdi+old } Q) \times (Q \sqsubseteq \text{fdi+old } P)$

$_ = \text{fdi}\infty\text{old} _ : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c)$
 $(Q : \text{Process}\infty \infty \{lu\} c) \rightarrow \text{Set}$
 $P = \text{fdi}\infty\text{old } Q = (P \sqsubseteq \text{fdi}\infty\text{old } Q) \times (Q \sqsubseteq \text{fdi}\infty\text{old } P)$

A.42 fdiRefusalPartsRemoved.agda

module fdiRefusalPartsRemoved where

open import process
 open import Size

```

open import choiceSetU
open import primitiveProcess
open import div
open import labelUniv
open import Data.Fin
open import Data.List
open import Data.Sum
open import TraceWithNextProcess
open import dataAuxFunction
open import Data.Bool.Base renaming (T to T')
open import Data.Unit
open import Data.Maybe
open import dataAuxFunction
open import Data.Empty
open import TraceWithoutSize
open import RefWithoutSize
open import auxData
open import fdi

-- in the following represent the refusels in schnieder book.

data refusalS $\infty$  {lu : LUniv}{c : Choice}(P : Process $\infty$   $\infty$  {lu} c)
  (X : (Label lu)  $\rightarrow$  Bool) : Set where
  refusalp : (Q : Process $\infty$  {lu} c)
    (tr : TrP $\infty$  {lu}{c} [] (inj1 Q) P)
    (stab : stable Q)
    (Xreject : (c : ChoiceSet (Ep Q))  $\rightarrow$   $\neg$  (T' (X (Labp Q c))))
     $\rightarrow$  refusalS $\infty$  P X

```

A.43 hidingOperator.agda

```

--@PREFIX@hiding

module hidingOperator where

open import Size
open import process
open import Data.Bool renaming (T to Tb)
open import Data.String.Base

```

```

open import auxData
open import dataAuxFunction
open import choiceSetU
open import Data.Sum
open import labelUniv

--@BEGIN@hidingDef

HideStr : {lu : LUniv}(f : Label lu → Bool) → String → String
HideStr f str = "Hide " ++ labelBoolFunToString f ++ " " ++ str

mutual
  Hide∞ : {i : Size}{lu : LUniv} → {c : Choice}
    → (hide : Label lu → Bool)
    → Process∞ i {lu} c
    → Process∞ i {lu} c
  forcep (Hide∞ f P) = Hide f (forcep P)
  Str∞ (Hide∞ f P) = HideStr f (Str∞ P)

  Hide : {i : Size} → {lu : LUniv} → {c : Choice}
    → (hide : Label lu → Bool)
    → Process i {lu} c
    → Process i {lu} c
  Hide f (node P) = node (Hide+ f P)
  Hide f (terminate x) = terminate x

  Hide+ : {i : Size} → {lu : LUniv} → {c : Choice}
    → (hide : Label lu → Bool) → Process+ i {lu} c
    → Process+ i {lu} c
  E (Hide+ f P) = subset' (E P) (¬b ∘ (f ∘ (Lab P)))
  Lab (Hide+ f P) c = Lab P (projSubset c)
  PE (Hide+ f P) c = Hide∞ f (PE P (projSubset c))
  I (Hide+ f P) = I P ⊕' subset' (E P) (f ∘ Lab P)
  PI (Hide+ f P) (inj1 c) = Hide∞ f (PI P c)
  PI (Hide+ f P) (inj2 c) = Hide∞ f (PE P (projSubset c))
  T (Hide+ f P) = T P
  PT (Hide+ f P) = PT P
  Str+ (Hide+ f P) = HideStr f (Str+ P)

```

--@END

mutual

HideWithName ∞ : {i : Size} → {c : Choice}
 → {lu : LUniv}
 → (name : String → String)
 → (hide : Label lu → Bool)
 → Process ∞ i {lu} c
 → Process ∞ i {lu} c

forcep (HideWithName ∞ name f P) = HideWithName name f (forcep P)
 Str ∞ (HideWithName ∞ name f P) = name (Str ∞ P)

HideWithName : {i : Size} → {c : Choice}
 → {lu : LUniv}
 → (name : String → String)
 → (hide : Label lu → Bool)
 → Process i {lu} c
 → Process i {lu} c

HideWithName name f (node P) = node (HideWithName+ name f P)
 HideWithName name f (terminate x) = terminate x

HideWithName+ : {i : Size} → {c : Choice} → {lu : LUniv}
 → (name : String → String)
 → (hide : Label lu → Bool) → Process+ i {lu} c
 → Process+ i {lu} c

E (HideWithName+ name f P) = subset' (E P) (\neg b \circ (f \circ (Lab P)))

Lab (HideWithName+ name f P) c = Lab P (projSubset c)

PE (HideWithName+ name f P) c = HideWithName ∞ name f (PE P (projSubset c))

I (HideWithName+ name f P) = I P \uplus ' subset' (E P) (f \circ Lab P)

PI (HideWithName+ name f P) (inj₁ c) = HideWithName ∞ name f (PI P c)

PI (HideWithName+ name f P) (inj₂ c) = HideWithName ∞ name f (PE P (projSubset c))

T (HideWithName+ name f P) = T P

PT (HideWithName+ name f P) = PT P

Str+ (HideWithName+ name f P) = name (Str+ P)

A.44 `interleave.agda`

```
--@PREFIX@interleaving
```

```
module interleave where
```

```
open import Size
open import process
open import choiceSetU
open import auxData
open import Data.Sum
open import Data.String renaming (_++_ to _++s_)
open import renamingResult
open import labelUniv
```

```
--@BEGIN@interleavingDef
```

```
_|||Str_ : String → String → String
s |||Str s' = s ++s "|||" ++s s'
```

```
mutual
```

```
_|||∞_ : {i : Size}{lu : LUniv}
        → {c₀ c₁ : Choice}
        → Process∞ i {lu} c₀
        → Process∞ i {lu} c₁
        → Process∞ i {lu} (c₀ ×' c₁)
forcep (P |||∞ Q) = forcep P ||| forcep Q
Str∞ (P |||∞ Q) = Str∞ P |||Str Str∞ Q
```

```
_|||_ : {i : Size}{lu : LUniv} → {c₀ c₁ : Choice} → Process i {lu} c₀
        → Process i c₁ → Process i {lu} (c₀ ×' c₁)
node P ||| node Q = node (P |||++ Q)
terminate a ||| Q = fmap (λ b → (a „ b)) Q
P ||| terminate b = fmap (λ a → (a „ b)) P
```

```
_|||∞+_ : {i : Size}{lu : LUniv} → {c₀ c₁ : Choice} → Process∞ i {lu} c₀
        → Process+ i {lu} c₁ → Process∞ i {lu} (c₀ ×' c₁)
```

$\text{forcep } (P \parallel_{\infty}^+ Q) = \text{node } (\text{forcep } P \parallel_{\text{p}}^+ Q)$
 $\text{Str}_{\infty} \quad (P \parallel_{\infty}^+ Q) = \text{Str}_{\infty} \quad P \parallel_{\text{Str}} \text{Str}^+ Q$

$_{\parallel}^+ \infty _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}^+ i \{lu\} \ c_0$
 $\rightarrow \text{Process}_{\infty} i \{lu\} \ c_1 \rightarrow \text{Process}_{\infty} i \{lu\} \ (c_0 \times' c_1)$
 $\text{forcep } (P \parallel_{\infty}^+ Q) = \text{node } (P \parallel_{\text{p}}^+ \text{forcep } Q)$
 $\text{Str}_{\infty} \quad (P \parallel_{\infty}^+ Q) = \text{Str}^+ \quad P \parallel_{\text{Str}} \text{Str}_{\infty} Q$

$_{\parallel}^+ \text{p} _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process} i \{lu\} \ c_0$
 $\rightarrow \text{Process}^+ i \{lu\} \ c_1 \rightarrow \text{Process}^+ i \{lu\} \ (c_0 \times' c_1)$
 $\text{terminate } a \parallel_{\text{p}}^+ Q = \text{fmap}^+ (\lambda b \rightarrow (a \text{ ,, } b)) Q$
 $\text{node } P \parallel_{\text{p}}^+ Q = P \parallel_{++}$

$_{\parallel}^+ \text{p} _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}^+ i \ c_0$
 $\rightarrow \text{Process} i \{lu\} \ c_1 \rightarrow \text{Process}^+ i \{lu\} \ (c_0 \times' c_1)$
 $P \parallel_{\text{p}}^+ \text{terminate } b = \text{fmap}^+ (\lambda a \rightarrow (a \text{ ,, } b)) P$
 $P \parallel_{\text{p}}^+ \text{node } Q = P \parallel_{++}$

$_{\parallel}^+ ++ _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\}$
 $\rightarrow \text{Process}^+ i \{lu\} \ c_0 \rightarrow \text{Process}^+ i \{lu\} \ c_1$
 $\rightarrow \text{Process}^+ i \{lu\} \ (c_0 \times' c_1)$

$\text{E } (P \parallel_{++} Q) = \text{E } P \uplus' \text{E } Q$
 $\text{Lab } (P \parallel_{++} Q) (\text{inj}_1 \ c) = \text{Lab } P \ c$
 $\text{Lab } (P \parallel_{++} Q) (\text{inj}_2 \ c) = \text{Lab } Q \ c$
 $\text{PE } (P \parallel_{++} Q) (\text{inj}_1 \ c) = \text{PE } P \ c \parallel_{\infty}^+ Q$
 $\text{PE } (P \parallel_{++} Q) (\text{inj}_2 \ c) = P \parallel_{\infty}^+ \text{PE } Q \ c$
 $\text{I } (P \parallel_{++} Q) = \text{I } P \uplus' \text{I } Q$
 $\text{PI } (P \parallel_{++} Q) (\text{inj}_1 \ c) = \text{PI } P \ c \parallel_{\infty}^+ Q$
 $\text{PI } (P \parallel_{++} Q) (\text{inj}_2 \ c) = P \parallel_{\infty}^+ \text{PI } Q \ c$
 $\text{T } (P \parallel_{++} Q) = \text{T } P \times' \text{T } Q$
 $\text{PT } (P \parallel_{++} Q) (c \text{ ,, } c_1) = \text{PT } P \ c \text{ ,, } \text{PT } Q \ c_1$
 $\text{Str}^+ (P \parallel_{++} Q) = \text{Str}^+ P \parallel_{\text{Str}} \text{Str}^+ Q$

$_{\parallel}^+ \text{p} \infty _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process} i \{lu\} \ c_0$
 $\rightarrow \text{Process}_{\infty} i \{lu\} \ c_1 \rightarrow \text{Process}_{\infty} i \{lu\} \ (c_0 \times' c_1)$
 $\text{forcep } (P \parallel_{\text{p}} \infty Q) = P \parallel_{\text{p}} \text{forcep } Q$
 $\text{Str}_{\infty} (P \parallel_{\text{p}} \infty Q) = \text{Str } P \parallel_{\text{Str}} \text{Str}_{\infty} Q$

$_{\parallel}^+ \infty \text{p} _ : \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}_{\infty} i \{lu\} \ c_0$
 $\rightarrow \text{Process} i \{lu\} \ c_1 \rightarrow \text{Process}_{\infty} i \{lu\} \ (c_0 \times' c_1)$

```

forcep (P |||∞p      Q) = forcep P ||| Q
Str∞ (P |||∞p Q)      = Str∞ P |||Str Str Q

```

```
--@END
```

```

infixl 10 _|||∞_
infixl 10 _|||_

```

```
mutual
```

```

_|||wNam∞_Using_,_,_ : {i : Size}
  → {lu : LUniv}
  → {c₀ c₁ : Choice}
  → Process∞ i {lu} c₀
  → Process∞ i {lu} c₁
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c₀ → String → String)
  → (fmapRightName : ChoiceSet c₁ → String → String)
  → Process∞ i {lu} (c₀ ×' c₁)
forcep (P |||wNam∞      Q Using ///name , fmapLeftName , fmapRightName)
  = forcep P      |||wNam forcep      Q Using ///name , fmapLeftName , fmapRightName
Str∞ (P |||wNam∞ Q Using ///name , fmapLeftName , fmapRightName)
  = ///name (Str∞      P) (Str∞ Q)

```

```

_|||wNam_Using_,_,_ : {i : Size}
  → {lu : LUniv}
  → {c₀ c₁ : Choice} → Process i {lu} c₀
  → Process i {lu} c₁
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c₀ → String → String)
  → (fmapRightName : ChoiceSet c₁ → String → String)
  → Process i {lu} (c₀ ×' c₁)
node P      |||wNam node Q Using ///name , fmapLeftName , fmapRightName
  = node (P |||wNam++ Q Using ///name , fmapLeftName , fmapRightName)
terminate a |||wNam      Q Using ///name , fmapLeftName , fmapRightName
  = fmapWithName (fmapLeftName a) (λ b → (a „ b)) Q
P |||wNam terminate b      Using ///name , fmapLeftName , fmapRightName
  = fmapWithName (fmapRightName b) (λ a → (a „ b)) P

```

```

-|||wNam $\infty$ +_Using_,_,_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process $\infty$  i {lu} c0
  → Process+ i {lu} c1
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c0 → String → String)
  → (fmapRightName : ChoiceSet c1 → String → String)
  → Process $\infty$  i {lu} (c0 ×' c1)
forcep (P |||wNam $\infty$ + Q Using ///name , fmapLeftName , fmapRightName)
  = node (forcep P |||wNamp+ Q Using ///name , fmapLeftName , fmapRightName)
Str $\infty$  (P |||wNam $\infty$ + Q Using ///name , fmapLeftName , fmapRightName)
  = ///name (Str $\infty$  P) (Str+ Q)

-|||wNam+ $\infty$ _Using_,_,_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process+ i {lu} c0
  → Process $\infty$  i {lu} c1
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c0 → String → String)
  → (fmapRightName : ChoiceSet c1 → String → String)
  → Process $\infty$  i {lu} (c0 ×' c1)
forcep (P |||wNam+ $\infty$  Q Using ///name , fmapLeftName , fmapRightName)
  = node (P |||wNam+p forcep Q Using ///name , fmapLeftName , fmapRightName)
Str $\infty$  (P |||wNam+ $\infty$  Q Using ///name , fmapLeftName , fmapRightName)
  = ///name (Str+ P) (Str $\infty$  Q)

-|||wNamp+_Using_,_,_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process i {lu} c0
  → Process+ i {lu} c1
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c0 → String → String)
  → (fmapRightName : ChoiceSet c1 → String → String)
  → Process+ i {lu} (c0 ×' c1)
terminate a |||wNamp+ Q Using ///name , fmapLeftName , fmapRightName
  = fmapWithName+ (fmapLeftName a) (λ b → (a ,, b)) Q
node P |||wNamp+ Q Using ///name , fmapLeftName , fmapRightName
  = P |||wNam++ Q Using ///name , fmapLeftName , fmapRightName

-|||wNam+p_Using_,_,_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process+ i {lu} c0
  → Process i {lu} c1
  → (///name : String → String → String)
  → (fmapLeftName : ChoiceSet c0 → String → String)
  → (fmapRightName : ChoiceSet c1 → String → String)
  → Process+ i {lu} (c0 ×' c1)
P |||wNam+p terminate b Using ///name , fmapLeftName , fmapRightName =

```

$$P \text{ |||wNam+p node } Q \quad \text{Using } \text{fmapWithName+ } (\text{fmapRightName } b) (\lambda a \rightarrow (a \text{ ,, } b)) P$$

$$\text{Using } \text{///name , fmapLeftName , fmapRightName} =$$

$$P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$_||\text{wNam++_Using_,-,-} : \{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\}$$

$$\rightarrow \text{Process+ } i \ \{lu\} \ c_0 \rightarrow \text{Process+ } i \ \{lu\} \ c_1$$

$$\rightarrow (\text{///name} : \text{String} \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow (\text{fmapLeftName} : \text{ChoiceSet } c_0 \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow (\text{fmapRightName} : \text{ChoiceSet } c_1 \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow \text{Process+ } i \ \{lu\} \ (c_0 \times' c_1)$$

$$\text{E } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) = \text{E } P \uplus'$$

$$\text{Lab } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_1 \ c) = \text{Lab } P \ c$$

$$\text{Lab } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_2 \ c) = \text{Lab } Q \ c$$

$$\text{PE } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_1 \ c) =$$

$$\text{PE } P \ c \text{ |||wNam}\infty+ \ Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$\text{PE } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_2 \ c) =$$

$$P \text{ |||wNam}\infty+ \ \text{PE } Q \ c \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$\text{I } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) = \text{I } P \uplus'$$

$$\text{PI } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_1 \ c) =$$

$$\text{PI } P \ c \text{ |||wNam}\infty+ \ Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$\text{PI } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (\text{inj}_2 \ c) =$$

$$P \text{ |||wNam}\infty+ \ \text{PI } Q \ c \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$\text{T } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) = \text{T } P \times'$$

$$\text{PT } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) (c \text{ ,, } c_1) = \text{PT } P \ c$$

$$\text{Str+ } (P \text{ |||wNam++ } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) = \text{///name}$$

$$_||\text{wNam}\infty_Using_,-,-} : \{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process } i \ \{lu\}$$

$$\rightarrow \text{Process}\infty \ i \ \{lu\} \ c_1$$

$$\rightarrow (\text{///name} : \text{String} \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow (\text{fmapLeftName} : \text{ChoiceSet } c_0 \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow (\text{fmapRightName} : \text{ChoiceSet } c_1 \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow \text{Process}\infty \ i \ \{lu\} \ (c_0 \times' c_1)$$

$$\text{forcep } (P \text{ |||wNam}\infty \ Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) =$$

$$P \text{ |||wNam forcep } Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}$$

$$\text{Str}\infty \ (P \text{ |||wNam}\infty \ Q \text{ Using } \text{///name , fmapLeftName , fmapRightName}) = \text{///name } (S$$

$$_||\text{wNam}\infty\text{p_Using_,-,-} : \{i : \text{Size}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \rightarrow \text{Process}\infty \ i$$

$$\rightarrow \text{Process } i \ \{lu\} \ c_1$$

$$\rightarrow (\text{///name} : \text{String} \rightarrow \text{String} \rightarrow \text{String})$$

$$\rightarrow (\text{fmapLeftName} : \text{ChoiceSet } c_0 \rightarrow \text{String} \rightarrow \text{String})$$



```

→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process∞ i {lu} (c₀ ×' c₁)
forcep (P |||wNam∞p      Q Using ///name , fmapLeftName , fmapRightName) =
      forcep P |||wNam Q Using ///name , fmapLeftName , fmapRightName
Str∞ (P |||wNam∞p      Q Using ///name , fmapLeftName , fmapRightName) = ///name (Str∞ P)

-- infixl 10 _|||wNam∞_
-- infixl 10 _|||wNam_

```

mutual

```

_|||w∞_ : {i : Size}
→ {lu : LUniv}
→ {c₀ c₁ : Choice}
→ Process∞ i {lu} c₀
→ Process∞ i {lu} c₁
→ {///name : String → String → String}
→ {fmapLeftName : ChoiceSet c₀ → String → String}
→ {fmapRightName : ChoiceSet c₁ → String → String}
→ Process∞ i {lu} (c₀ ×' c₁)
_|||w∞_ {i} {lu} {c} {c₀} P Q {///name}{fmapLeftName}{fmapRightName} = P |||wNam∞ Q Us

_|||w_ : {i : Size}
→ {lu : LUniv}
→ {c₀ c₁ : Choice}
→ Process i {lu} c₀
→ Process i {lu} c₁
→ {///name : String → String → String}
→ {fmapLeftName : ChoiceSet c₀ → String → String}
→ {fmapRightName : ChoiceSet c₁ → String → String}
→ Process i {lu} (c₀ ×' c₁)
_|||w_ {i} {lu} {c} {c₀} P Q {///name}{fmapLeftName}{fmapRightName} = P |||wNam Q Using /

```



```

-|||w+_- : {i : Size}
  → {lu : LUniv}
  → {c0 c1 : Choice}
  → Process+ i {lu} c0
  → Process+ i {lu} c1
  → {///name : String → String → String}
  → {fmapLeftName : ChoiceSet c0 → String → String}
  → {fmapRightName : ChoiceSet c1 → String → String}
  → Process+ i {lu} (c0 ×' c1)
-|||w+_- {i} {lu} {c} {c0} P Q {///name}{fmapLeftName}{fmapRightName} = P |||wNam

```

A.45 internalChoice.agda

```
--@PREFIX@internalChoice
```

```
module internalChoice where
```

```

open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List.Base renaming (map to mapL)
open import Data.Fin
open import Size
open import choiceSetU
open import process
open import showFunction
open import dataAuxFunction
open import labelUniv

```

```

bool : Choice
bool = fin 2

```

```

if_then_else : {A : Set} → ChoiceSet bool → A → A → A
if zero then a else b = a
if (suc zero) then a else b = b
if (suc (suc ())) then a else b

```

```
--@BEGIN@intDef
```

```
⊔Str_ : String → String → String
s ⊔Str s' = "(" ++s s ++s " ⊔ " ++s s' ++s ")"
```

```
⊔+_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process+ i {lu} c
E   (P ⊔+ Q)           = ∅'
Lab (P ⊔+ Q) ()
PE  (P ⊔+ Q) ()
I   (P ⊔+ Q)           = fin 2
PI  (P ⊔+ Q) zero      = P
PI  (P ⊔+ Q) (suc zero) = Q
PI  (P ⊔+ Q) (suc (suc ()))
T   (P ⊔+ Q)           = ∅'
PT  (P ⊔+ Q) ()
Str+ (P ⊔+ Q)          = Str∞ P ⊔Str Str∞ Q
```

```
⊔_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process i {lu} c
P ⊔ Q = node (P ⊔+ Q)
```

```
⊔∞_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process∞ (↑ i) {lu} c
forcep (P ⊔∞ Q) {j} = P ⊔ Q
Str∞ (P ⊔∞ Q) = (Str∞ P) ⊔Str (Str∞ Q)
```

```
--@END
```

```
⊔+'_ : {i : Size} → {c : Choice} → {lu : LUniv} → Process∞ i {lu} c
      → Process∞ i {lu} c → Process+ i {lu} c
P ⊔+' Q = process+ ∅' efq efq bool (λ b → if b then P else Q) ∅' efq
      (Str∞ P ⊔Str Str∞ Q)
```

```
mutual
```

```

IntChoiceStr : {c : Choice} → (f : ChoiceSet c → String) → String
IntChoiceStr f = " \t □ \t " ++s choice2Str2Str f

IntChoice∞ : (i : Size) → {c₀ : Choice} → (c : Choice) → {lu : LUniv}
→ (PI : ChoiceSet c → Process∞ i {lu} c₀)
→ Process∞ (↑ i) {lu} c₀
forcep (IntChoice∞ i c {lu} PI) {j} = IntChoice j c {lu} PI
Str∞ (IntChoice∞ i c PI) = IntChoiceStr (Str∞ ∘ PI)

IntChoice : (i : Size) → {c₀ : Choice} → (c : Choice) → {lu : LUniv}
→ (PI : ChoiceSet c → Process∞ i {lu} c₀)
→ Process i {lu} c₀
IntChoice i c PI = node (IntChoice+ i c PI)

IntChoice+ : (i : Size) → {c₀ : Choice} → (c : Choice) → {lu : LUniv}
→ (P : ChoiceSet c → Process∞ i {lu} c₀)
→ Process+ i {lu} c₀
E      (IntChoice+ i c P) = ∅'
Lab    (IntChoice+ i c P) = efq
PE     (IntChoice+ i c P) = efq
I      (IntChoice+ i c P) = c
PI     (IntChoice+ i c P) = P
T      (IntChoice+ i c P) = ∅'
PT     (IntChoice+ i c P) = efq
Str+   (IntChoice+ i c P) = IntChoiceStr (Str∞ ∘ P)

--@END

```

A.46 interrupt.agda

```

--@PREFIX@interrupting

module interrupt where

open import Size
open import process
open import choiceSetU
open import Data.String renaming (_++_ to _++s_)
open import renamingResult

```

```

open import Data.Sum
open import addTick
open import labelUniv

```

```

_ΔRes_ : Choice → Choice → Choice
c₀ ΔRes c₁ = (c₀ ⊕' c₁) ⊕' (c₀ ×' c₁)

```

```

_ΔStr_ : String → String → String
s ΔStr s' = s ++s " \t Δ \t " ++s s'

```

```

--@BEGIN@interruptDef

```

```

mutual

```

```

_Δ∞∞_ : {lu : LUniv}{c₀ c₁ : Choice} → {i : Size}
        → Process∞ i {lu} c₀ → Process∞ i {lu} c₁
        → Process∞ i {lu} (c₀ ⊕' c₁)
forcep (P Δ∞∞ P') = forcep P Δ forcep P'
Str∞      (P Δ∞∞ P') = Str∞ P ΔStr Str∞ P'

```

```

_Δ_ : {lu : LUniv}{c₀ c₁ : Choice} → {i : Size}
      → Process i {lu} c₀ → Process i {lu} c₁
      → Process i {lu} (c₀ ⊕' c₁)
node P Δ P' = P Δ+p P'
P Δ node P' = P Δp+ P'
terminate a Δ terminate b = 2-✓ a b

```

```

_Δ+_ : {lu : LUniv}{c₀ c₁ : Choice} → {i : Size}
      → Process+ i {lu} c₀ → Process+ i {lu} c₁
      → Process+ i {lu} (c₀ ⊕' c₁)
E (P Δ+ Q) = E P ⊕' E Q
Lab (P Δ+ Q) (inj₁ x) = Lab P x
Lab (P Δ+ Q) (inj₂ x) = Lab Q x
PE (P Δ+ Q) (inj₁ x) = PE P x Δ∞+ Q
PE (P Δ+ Q) (inj₂ x) = fmap∞ inj₂ (PE Q x)

```

$I \quad (P \triangle+ Q) = I \ P \ \uplus' \ I \ Q$
 $PI \quad (P \triangle+ Q) \ (inj_1 \ c) = PI \ P \ c \ \triangle\infty+ \ Q$
 $PI \quad (P \triangle+ Q) \ (inj_2 \ c) = P \ \triangle+\infty \ PI \ Q \ c$
 $T \quad (P \triangle+ Q) = T \ P \ \uplus' \ T \ Q$
 $PT \quad (P \triangle+ Q) \ (inj_1 \ c) = inj_1 \ (PT \ P \ c)$
 $PT \quad (P \triangle+ Q) \ (inj_2 \ c) = inj_2 \ (PT \ Q \ c)$
 $Str+ \quad (P \triangle+ Q) = Str+ \ P \ \triangle Str \ Str+ \ Q$

--@END

$_ \triangle\infty+ _ : \{lu : LUniv\} \{c_0 \ c_1 : Choice\} \rightarrow \{i : Size\}$
 $\rightarrow Process\infty \ i \ \{lu\} \ c_0 \rightarrow Process+ \ i \ \{lu\} \ c_1$
 $\rightarrow Process\infty \ i \ \{lu\} \ (c_0 \ \uplus' \ c_1)$
 $forcep \ (P \ \triangle\infty+ \ P') = forcep \ P \ \triangle p+ \ P'$
 $Str\infty \ (P \ \triangle\infty+ \ P') = Str\infty \ P \ \triangle Str \ Str+ \ P'$

$_ \triangle+\infty _ : \{lu : LUniv\} \{c_0 \ c_1 : Choice\} \rightarrow \{i : Size\}$
 $\rightarrow Process+ \ i \ \{lu\} \ c_0 \rightarrow Process\infty \ i \ \{lu\} \ c_1$
 $\rightarrow Process\infty \ i \ \{lu\} \ (c_0 \ \uplus' \ c_1)$
 $forcep \ (P \ \triangle+\infty \ P') = P \ \triangle+p \ forcep \ P'$
 $Str\infty \ (P \ \triangle+\infty \ P') = Str+ \ P \ \triangle Str \ Str\infty \ P'$

$_ \triangle+p _ : \{lu : LUniv\} \{c_0 \ c_1 : Choice\} \rightarrow \{i : Size\}$
 $\rightarrow Process+ \ i \ \{lu\} \ c_0 \rightarrow Process \ i \ \{lu\} \ c_1$
 $\rightarrow Process \ i \ \{lu\} \ (c_0 \ \uplus' \ c_1)$
 $P \ \triangle+p \ \text{terminate } b = add\checkmark \ (inj_2 \ b)$
 $\quad \quad \quad (node \ (fmap+ \ inj_1 \ P) \)$
 $P \ \triangle+p \ \text{node } P' = node \ (P \ \triangle+ \ P')$

$_ \triangle p+ _ : \{lu : LUniv\} \{c_0 \ c_1 : Choice\} \rightarrow \{i : Size\}$
 $\rightarrow Process \ i \ \{lu\} \ c_0 \rightarrow Process+ \ i \ \{lu\} \ c_1$
 $\rightarrow Process \ i \ \{lu\} \ (c_0 \ \uplus' \ c_1)$
 $terminate \ a \ \triangle p+ \ P' = addTimed\checkmark \ (inj_1 \ a)$
 $\quad \quad \quad (node \ (fmap+ \ inj_2 \ P'))$
 $node \ P \ \triangle p+ \ P' = node \ (P \ \triangle+ \ P')$

A.47 IOExampleScreenShotForTyDePaper2.agda

```

module IOExampleScreenShotForTyDePaper2 where

open import Size
open import process
open import choiceSetU
open import prefix
open import primitiveProcess
open import auxData
open import simulator
open import Data.Fin renaming (_+_ to _+_,_<_ to _<F_)
open import Data.List
open import SizedIO.Console hiding (main)
open import NativeIO
open import externalChoice
open import interleave
open import internalChoice
open import UnitModule
open import labelUniv
open import label renaming (Label to LabelSimple)
open import Data.Bool

setSTOP : Choice
setSTOP = namedElements ("STOP" :: [])

skip :  $\forall \{i\} \rightarrow \text{Process}_\infty i \{ \text{Isimple} \} \text{setSTOP}$ 
skip = delay (SKIP (ne zero))

process0 :  $\forall i \rightarrow \text{Process } i \{ \text{Isimple} \} \text{setSTOP}$ 
process0  $i$  = lab laba  $\longrightarrow \text{STOP}_\infty \text{setSTOP}$ 

process1 :  $\forall i \rightarrow \text{Process } i \{ \text{Isimple} \} \text{setSTOP}$ 
process1  $i$  = lab labb  $\longrightarrow_{\text{pp}} (\text{lab laba} \longrightarrow \text{STOP}_\infty \text{setSTOP})$ 

process2 :  $\forall i \rightarrow \text{Process } i \{ \text{Isimple} \} \text{setSTOP}$ 
process2  $i$  = lab labc  $\longrightarrow \text{STOP}_\infty \text{setSTOP}$ 

process3 :  $\forall i \rightarrow \text{Process } i \{ \text{Isimple} \} \text{setSTOP}$ 

```

```

process3 i = lab laba → STOP∞ setSTOP

process4 : ∀ i → Process i {Isimple} setSTOP
process4 i = delay (process2 i) □ delay (process3 i)

transition2 : ∀ i → Process i {Isimple} setSTOP
transition2 i = lab laba → delay {i} (lab labb → delay {↑ i} (lab labc → delay {↑ (↑ i)} (S

process5 : ∀ i → Process i {Isimple} (setSTOP ⊕' (setSTOP ⊕' setSTOP))
process5 i = process1 i □ (process4 i □ SKIP (ne zero))

--need to fix

main : NativeIO Unit
main = translatoConsole (myProgram true (setSTOP ⊕' (setSTOP ⊕' setSTOP)) (process5 ○

```

A.48 IOSeqCom.agda

```

module IOSeqCom where

open import Data.Bool
open import Size
open import process
open import choiceSetU
open import prefix
open import primitiveProcess
open import label
open import sequentialComposition
open import simulator
open import Data.Fin renaming (_+_ to _+,_<_ to _<F_)
open import Data.List
open import SizedIO.Console hiding (main)
open import NativeIO
open import UnitModule
open import labelUniv
open import label renaming (Label to LabelSimple)

```

```

setSTOP : Choice

```

```

setSTOP = namedElements ("STOP" :: [])

SetSTOP : Set
SetSTOP = ChoiceSet setSTOP

setAB : Choice
setAB = namedElements ("A" :: "B" :: [])

SetAB : Set
SetAB = ChoiceSet setAB

transition1 : ∀ i → Process i {lsimple} setSTOP
transition1 i = lab laba → delay {i} (lab labb → delay {↑ i} (lab labc → delay {↑ (↑ i)} (STOP set

transition2 : ∀ i → Process i {lsimple} setSTOP
transition2 i = lab laba → delay {i} (lab labb → delay {↑ i} (lab labc → delay {↑ (↑ i)} (STOP set

transition3 : ∀ i → Process i {lsimple} setSTOP
transition3 i = transition1 i >>= (λ x → delay {i} (transition2 i))

main : NativeIO Unit
main = translatoConsole (myProgram true (namedElements ("STOP" :: [])) (transition3 ∞))

```

A.49 label.agda

```

--@PREFIX@label
module label where

--@BEGIN@Labelsimple

data Label : Set where
  laba labb labc : Label

--@END

```

A.50 labelEq.agda

```

module labelEq where

open import label
open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import Data.Unit

_==l_ : Label → Label → Bool
laba ==l laba = true
labb ==l labb = true
labc ==l labc = true
_ ==l _ = false

refl==l : {l : Label} → T' (l ==l l)
refl==l {laba} = tt
refl==l {labb} = tt
refl==l {labc} = tt

_==L_ : Label → Label → Set
l ==L l' = T' (l ==l l')
```

A.51 labelUniv.agda

```

--@PREFIX@labelUniv
module labelUniv where

open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import Data.String renaming (_++_ to _++s_)
open import Data.String.Base
open import Data.List.Base
open import Data.List
open import label renaming (Label to LabelSimple)
open import showLabelP hiding( labelBoolFunToString; labelLabelFunToString )
                        renaming(showLabel to showLabelSimple ;
```

```

                                LabelList to LabelListSimple )
open import labelEq   hiding ( _==L_ )
                      renaming ( _==l_ to _==lsimpl_ ;
                                refl==l to refl==lsimple )

-- Labelf stands for Labelfield similarly for _==lf_ etc

--@BEGIN@LUniv

record LUniv : Set1 where
  field
    Labelf      :      Set
    _==lf_      :      Labelf → Labelf → Bool
    refl==lf    :      {l : Labelf} → T' (l ==lf l)
    sym==lf     :      {l l' : Labelf} → T' (l ==lf l') → T' (l' ==lf l)
    transf      :      {l l' : Labelf} → (Q : Labelf → Set)
                      → T' (l ==lf l') → Q l → Q l'

    showLabelf  :      Labelf → String
    LabelListf  :      List Labelf

--@END
open LUniv public

mutual

--@BEGIN@Label1

data Label (lu : LUniv) : Set where
  lab : LUniv.Labelf lu → Label lu

--@END

--@BEGIN@EqLabel1

_==l_ : {lu : LUniv} → Label lu → Label lu → Bool
_==l_ {lu} (lab x) (lab y) = LUniv._==lf_ lu x y

```



```

refl==l : {lu : LUniv} {l : Label lu} → T' (l ==l l)
refl==l {lu} {lab x} = LUniv.refl==lf lu {x}

sym==l : {lu : LUniv} {l l' : Label lu} → T' (l ==l l') → T' (l' ==l l)
sym==l {lu} {lab x} {lab y} p = LUniv.sym==lf lu {x} {y} p

transfLu : {lu : LUniv} (Q : Label lu → Set) {l l' : Label lu}
  → T' (l ==l l')          → Q l → Q l'
transfLu {lu} Q {lab l} {lab l'} ll' q =
  LUniv.transf lu {l} {l'} (λ x → Q (lab x)) ll' q

_==L_ : {lu : LUniv} → Label lu → Label lu → Set
_==L_ {lu} l l' = T' (_==l_ {lu} l l')

showLabel : {lu : LUniv} → Label lu → String
showLabel {lu} (lab x) = LUniv.showLabelf lu x

LabelList : (lu : LUniv) → List (Label lu)
LabelList lu = map (λ x → lab x) (LUniv.LabelListf lu)

labelBoolFunToString : {lu : LUniv} → (Label lu → Bool) → String
labelBoolFunToString {lu} f = unlines (map (showLabel {lu})
  (filter f (LabelList lu)))

labelLabelFunToString : {lu : LUniv} → (Label lu → Label lu) → String
labelLabelFunToString {lu} f = "[["
  ++s unlinesWithChosenString ", "
    (map (λ l → showLabel {lu} (f l)
  ++s " <- " ++s showLabel {lu} l)
    (LabelList lu))
  ++s "]]"

--@END
open LUniv

--@BEGIN@trl

```



```

trl : {l l' : LabelSimple} → (Q : LabelSimple → Set)
    → T' (l ==lsimpl l') → Q l → Q l'
trl {laba} {laba} Q t q = q
trl {laba} {labbb} Q () q
trl {laba} {labcb} Q () q
trl {labbb} {laba} Q () q
trl {labbb} {labbb} Q t q = q
trl {labbb} {labcb} Q () q
trl {labcb} {laba} Q () q
trl {labcb} {labbb} Q () q
trl {labcb} {labcb} Q t q = q

symLabelSimple : {l l' : LabelSimple} → T' (l ==lsimpl l')
    → T' (l' ==lsimpl l)
symLabelSimple {laba} {laba} tt = tt
symLabelSimple {laba} {labbb} ()
symLabelSimple {laba} {labcb} ()
symLabelSimple {labbb} {laba} ()
symLabelSimple {labbb} {labbb} tt = tt
symLabelSimple {labbb} {labcb} ()
symLabelSimple {labcb} {laba} ()
symLabelSimple {labcb} {labbb} ()
symLabelSimple {labcb} {labcb} tt = tt

lsimple : LUniv
Labelf lsimple = LabelSimple
_==lf_ lsimple = _==lsimpl_
refl==lf lsimple {l} = refl==lsimple {l}
showLabelf lsimple = showLabelSimple
LabelListf lsimple = LabelListSimple
transf lsimple = trl
sym==lf lsimple {l} {l'} = symLabelSimple {l} {l'}

--@END

```

A.52 labelUnivAsPureRecord.agda

```
module labelUnivAsPureRecord where
```

```

open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import Data.String renaming (_++_ to _++s_)
open import Data.String.Base
open import Data.List.Base
open import Data.List
open import label renaming (Label to LabelSimple)
open import showLabelP hiding( labelBoolFunToString;labelLabelFunToString)
                        renaming(showLabel to showLabelSimple ;
                                LabelList to LabelListSimple )
open import labelEq hiding (_==L_)
                        renaming (_==l_ to _==lsimpl_ ;
                                refl==l to refl==lsimple )

```

```

mutual
  record LUniv : Set1 where
    field
      Labelf : Set
      _==lf_ : Labelf → Labelf → Bool
      refl==lf : {l : Labelf} → T' (l ==lf l)
      showLabelf : Labelf → String
      LabelListf : List Labelf

```

```

Label : LUniv -> Set
Label = LUniv.Labelf

```

```

_==l_ : {lu : LUniv} → Label lu → Label lu → Bool
_==l_ {lu} = LUniv._==lf_ lu

```

```

refl==l : {lu : LUniv} {l : Label lu} → T' (l ==l l)
refl==l {lu} = LUniv.refl==lf lu

```

```

_==L_ : {lu : LUniv} → Label lu → Label lu → Set
_==L_ {lu} l l' = T' (_==l_ {lu} l l')

```

```

showLabel : {lu : LUniv} → Label lu → String
showLabel {lu} = LUniv.showLabelf lu

```

```

LabelList : (lu : LUniv) → List (Label lu)
LabelList = LUniv.LabelListf

```

```

labelBoolFunToString : {lu : LUniv} → (Label lu → Bool) → String
labelBoolFunToString {lu} f = unlines (map (showLabel {lu}) (filter f (LabelList lu)))

```

```

labelLabelFunToString : {lu : LUniv} → (Label lu → Label lu) → String
labelLabelFunToString {lu} f = "[["
    ++s unlinesWithChosenString ", " (map (λ l → showLabel {lu} (f l)
    ++s " <- " ++s showLabel {lu} l) (LabelList lu))
    ++s "]"

```

```

open LUniv

```

```

Isimple : LUniv
Labelf Isimple = LabelSimple
_==lf_ Isimple = _==lsimpl_
refl==lf Isimple {l} = refl==lsimple {l}
showLabelf Isimple = showLabelSimple
LabelListf Isimple = LabelListSimple

```

A.53 labelUnivAsUniverse.agda

```

module labelUnivAsUniverse where

```

```

open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import Data.String renaming (_++_ to _++s_)
open import Data.String.Base
open import Data.List.Base

```

```

open import Data.List
open import label renaming (Label to LabelSimple)
open import showLabelP hiding( labelBoolFunToString ; labelLabelFunToString)
                        renaming(showLabel to showLabelSimple ;
                                LabelList to LabelListSimple )
open import labelEq hiding ( _==L_ )
                        renaming ( _==l_ to _==lsimpl_ ;
                                refl==l to refl==lsimple )

mutual
  data LUniv : Set where
    lsimple : LUniv

Label : LUniv -> Set
Label lsimple = LabelSimple

_==l_ : {lu : LUniv} → Label lu → Label lu → Bool
_==l_ {lsimple} = _==lsimpl_

refl==l : {lu : LUniv} {l : Label lu} → T' (l ==l l)
refl==l {lsimple} {l} = refl==lsimple {l}

_==L_ : {lu : LUniv} → Label lu → Label lu → Set
l ==L l' = T' (l ==l l')

showLabel : {lu : LUniv} → Label lu → String
showLabel {lsimple} = showLabelSimple

LabelList : (lu : LUniv) → List (Label lu)
LabelList lsimple = LabelListSimple

labelBoolFunToString : {lu : LUniv} → (Label lu → Bool) → String
labelBoolFunToString {lu} f = unlines (map (showLabel {lu}) (filter f (LabelList lu)))

```

```

labelLabelFunToString : {lu : LUniv} → (Label lu → Label lu) → String
labelLabelFunToString {lu} f = "[["
    ++s unlinesWithChosenString ", " (map (λ l → showLabel {lu} (f l)
    ++s " <- " ++s showLabel {lu} l) (LabelList lu))
    ++s "]" ]"

```

A.54 lemFmap.agda

```

module lemFmap where

```

```

open import process
open import Size
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.List
open import Data.Sum
open import labelUniv
open import RefWithoutSize
open import dataAuxFunction
open import renamingResult
open import TraceWithoutSize
open import addTick
open import Data.Fin
open import internalChoice

```

```

swap⊕ : {c₀ c₁ : Choice} → ChoiceSet (c₀ ⊕' c₁) → ChoiceSet (c₁ ⊕' c₀)
swap⊕ (inj₁ x) = inj₂ x
swap⊕ (inj₂ y) = inj₁ y

```

```

Ass⊕ : {c₀ c₁ c₂ : Choice} → ChoiceSet ((c₀ ⊕' c₁) ⊕' c₂) → ChoiceSet (c₀ ⊕' (c₁ ⊕' c₂))
Ass⊕ (inj₁ (inj₁ x)) = inj₁ x
Ass⊕ (inj₁ (inj₂ x)) = inj₂ (inj₁ x)
Ass⊕ (inj₂ x) = inj₂ (inj₂ x)

```

```

Ass⊕r : {c₀ c₁ c₂ : Choice} → ChoiceSet (c₀ ⊕' (c₁ ⊕' c₂)) → ChoiceSet ((c₀ ⊕' c₁) ⊕' c₂)

```

```

Ass⊔r (inj1 x) = inj1 (inj1 x)
Ass⊔r (inj2 (inj1 x)) = inj1 (inj2 x)
Ass⊔r (inj2 (inj2 x)) = inj2 x

```

```

swap× : {c0 c1 : Choice} → ChoiceSet (c0 ×' c1) → ChoiceSet (c1 ×' c0)
swap× (x0 ,, x1) = (x1 ,, x0)

```

```

Ass× : {c0 c1 c2 : Choice} → ChoiceSet (c0 ×' (c1 ×' c2)) → ChoiceSet ((c0 ×' c1)
Ass× (x ,, (x1 ,, x2)) = ((x ,, x1) ,, x2)

```

```

Ass×' : {c0 c1 c2 : Choice} → ChoiceSet ((c0 ×' c1) ×' c2) → ChoiceSet (c0 ×' (c1 ×' c2))
Ass×' ((x ,, x1) ,, x2) = x ,, (x1 ,, x2)

```

mutual

```

Fmap+ : {lu : LUniv}{c0 c1 c2 c3 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
→ (g : ChoiceSet c1 → ChoiceSet c2)
→ (h : ChoiceSet c2 → ChoiceSet c3)
→ (P : Process+ ∞ {lu} c0)
→ (l : List (Label lu))
→ (m : Maybe (ChoiceSet c3))
→ (tr : Tr {lu} l m (node (fmap+ h (fmap+ g (fmap+ f P)))))
→ Tr {lu} l m (node (fmap+ (h ∘ g ∘ f) P))
Fmap+ {lu} {c0} {c1} {c2} {c3} f g h P l m (tnode tr) = tnode (Fmap+ {lu} {c0} {c1} {c2} {c3} f g h P l m (tnode tr))

```

```

Fmap+ : {lu : LUniv}{c0 c1 c2 c3 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
→ (g : ChoiceSet c1 → ChoiceSet c2)
→ (h : ChoiceSet c2 → ChoiceSet c3)
→ (P : Process+ ∞ {lu} c0)
→ (fmap+ ((h ∘ (g ∘ f))) P) ⊑+ (fmap+ h (fmap+ g (fmap+ f P)))

```

```

Fmap+ f g h P .[] .nothing empty = empty

```

```

Fmap+ f g h P .(Lab P x :: l) m (extc l .m x x1) = extc l m x (Fmap∞ f g h (PE P x) l m x1)

```

```

Fmap+ f g h P l m (intc .l .m x x1) = intc l m x (Fmap∞ f g h (PI P x) l m x1)

```

```

Fmap+ f g h P .[] .(just (h (g (f (PT P x))))) (terc x) = terc x

```

```

Fmap∞ : {lu : LUniv}{c0 c1 c2 c3 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
→ (g : ChoiceSet c1 → ChoiceSet c2)

```

```

→ (h : ChoiceSet c₂ → ChoiceSet c₃)
→ (P : Process ∞ {lu} c₀)
→ (fmap∞ ((h ∘ (g ∘ f))) P) ⊆∞ (fmap∞ h (fmap∞ g (fmap∞ f P)))
Fmap∞ f g h P l m x = Fmap f g h (forcep P) l m x

```

```

Fmap : {lu : LUniv}{c₀ c₁ c₂ c₃ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
→ (g : ChoiceSet c₁ → ChoiceSet c₂)
→ (h : ChoiceSet c₂ → ChoiceSet c₃)
→ (P : Process ∞ {lu} c₀)
→ (fmap ((h ∘ (g ∘ f))) P) ⊆ (fmap h (fmap g (fmap f P)))
Fmap f g h (terminate x) l m x₁ = x₁
Fmap f g h (node x) l m x₁ = Fmap+ f g h x l m x₁

```

mutual

```

lemFmap+ : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
→ (g : ChoiceSet c₁ → ChoiceSet c₂)
→ (P : Process+ ∞ {lu} c₀)
→ (fmap+ (g ∘ f) P) ⊆+ (fmap+ g (fmap+ f P))

```

```

lemFmap+ f g P .[] .nothing empty = empty

```

```

lemFmap+ {lu} f g P .(Lab {∞} {lu} P x :: l) m (extc l .m x x₁) = extc l m x (lemFmap∞ f g (PE l .m x x₁))

```

```

lemFmap+ f g P l m (intc l .m x x₁) = intc l m x (lemFmap∞ f g (PI P x) l m x₁)

```

```

lemFmap+ f g P .[] .(just (g (f (PT P x)))) (terc x) = terc x

```

```

lemFmap∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
→ (g : ChoiceSet c₁ → ChoiceSet c₂)
→ (P : Process∞ ∞ {lu} c₀)
→ (fmap∞ (g ∘ f) P) ⊆∞ (fmap∞ g (fmap∞ f P))

```

```

lemFmap∞ f g P l m x = lemFmap f g (forcep P) l m x

```

```

lemFmap : {lu : LUniv}{c₀ c₁ c₂ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁)
→ (g : ChoiceSet c₁ → ChoiceSet c₂)
→ (P : Process ∞ {lu} c₀)
→ (fmap (g ∘ f) P) ⊆ (fmap g (fmap f P))

```

```

lemFmap f g (terminate x) l m x₁ = x₁

```

```

lemFmap f g (node P) l m (tnode x) = tnode (lemFmap+ f g P l m x)

```

mutual

```

lemFmap+R : {lu : LUniv}{c0 c1 c2 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
  → (g : ChoiceSet c1 → ChoiceSet c2)
  → (P : Process+ ∞ {lu} c0)
  → (fmap+ g (fmap+ f P)) ⊑+ (fmap+ (g ∘ f) P)
lemFmap+R f g P .[] .nothing empty = empty
lemFmap+R f g P .(Lab P x :: l) m (extc l .m x x1) = extc l m x (lemFmap∞R f g (PE P x))
lemFmap+R f g P l m (intc l .m x x1) = intc l m x (lemFmap∞R f g (PI P x) l m x1)
lemFmap+R f g P .[] .(just (g (f (PT P x)))) (terc x) = terc x

```

```

lemFmap∞R : {lu : LUniv}{c0 c1 c2 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
  → (g : ChoiceSet c1 → ChoiceSet c2)
  → (P : Process∞ ∞ {lu} c0)
  → (fmap∞ g (fmap∞ f P)) ⊑∞ (fmap∞ (g ∘ f) P)
lemFmap∞R f g P l m x = lemFmapR f g (forcep P) l m x

```

```

lemFmapR : {lu : LUniv}{c0 c1 c2 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
  → (g : ChoiceSet c1 → ChoiceSet c2)
  → (P : Process ∞ {lu} c0)
  → (fmap g (fmap f P)) ⊑ (fmap (g ∘ f) P)
lemFmapR f g (terminate x) l m x1 = x1
lemFmapR f g (node P) l m (tnode x) = tnode (lemFmap+R f g P l m x)

```

mutual

```

addTimeFmapLemma+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (f : ChoiceSet c0 → ChoiceSet c1)
  (g : ChoiceSet c1 → ChoiceSet c2)
  (P : Process+ ∞ {lu} c0)
  (a : ChoiceSet c1)
  → addTimed✓+ (g a) (fmap+ (g ∘ f) P) ⊑+ fmap+ g (addTimeFmapLemma+
addTimeFmapLemma+ {c0} {c1} {c2} f g P a .[] .nothing empty = empty
addTimeFmapLemma+ {c0} {c1} {c2} f g P a .(Lab P x :: l) m (extc l .m x q) = extc l m x (addTimeFmapLemma+
addTimeFmapLemma+ {c0} {c1} {c2} f g P a l m (intc l .m x q) = intc l m x (addTimeFmapLemma+
addTimeFmapLemma+ {c0} {c1} {c2} f g P a .[] .(just (g a)) (terc (inj1 x)) = terc (inj1 x)
addTimeFmapLemma+ {c0} {c1} {c2} f g P a .[] .(just (g (f (PT P y)))) (terc (inj2 y)) =

```

```

addTimeFmapLemma∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process∞ ∞ {lu} c₀)
  (a : ChoiceSet c₁)
  → addTimed✓∞ (g a) (fmap∞ (g ∘ f) P) ⊆∞ fmap∞ g (addTimed✓∞
addTimeFmapLemma∞ {c₀} {c₁} {c₂} f g P a l m q = addTimeFmapLemma f g (forceP P) a l m

```

```

addTimeFmapLemma : {lu : LUniv}
  {c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process ∞ {lu} c₀)
  (a : ChoiceSet c₁)
  → addTimed✓ (g a) (fmap (g ∘ f) P) ⊆ fmap g (addTimed✓ a (fmap f P))
addTimeFmapLemma f g (terminate x) a .[] .nothing (tnode empty) = tnode empty
addTimeFmapLemma f g (terminate x) a .(Lab (2-✓+ a (f x)) - :: l₁) m (tnode (extc l₁ .m () tr))
addTimeFmapLemma f g (terminate x) a l m (tnode (intc .l .m () tr))
addTimeFmapLemma f g (terminate x) a .[] .(just (g a)) (tnode (terc zero)) = tnode (terc zero)
addTimeFmapLemma f g (terminate x) a .[] .(just (g (f x))) (tnode (terc (suc zero))) = tnode (terc (suc zero))
addTimeFmapLemma f g (terminate x) a .[] .(just (g (unifyA⊕A (PT (2-✓+ a (f x)) (suc (suc -))))))
addTimeFmapLemma f g (node P) a l m (tnode tr) = tnode (addTimeFmapLemma+ f g P a l m tr)

```

mutual

```

addTimeFmapLemma+R : {lu : LUniv}{c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process+ ∞ {lu} c₀)
  (a : ChoiceSet c₁)
  → fmap+ g (addTimed✓+ a (fmap+ f P)) ⊆+ addTimed✓+ (g a) (fmap+ f P)
addTimeFmapLemma+R {c₀} {c₁} {c₂} f g P a .[] .nothing empty = empty
addTimeFmapLemma+R {c₀} {c₁} {c₂} f g P a .(Lab P x :: l) m (extc l .m x q) = extc l m x (lemFmap+R f g P a l m q)
addTimeFmapLemma+R {c₀} {c₁} {c₂} f g P a l m (intc .l .m x q) = intc l m x (addTimeFmapLemma+R f g P a l m q)
addTimeFmapLemma+R {c₀} {c₁} {c₂} f g P a .[] .(just (g a)) (terc (inj₁ x)) = terc (inj₁ x)
addTimeFmapLemma+R {c₀} {c₁} {c₂} f g P a .[] .(just (g (f (PT P y)))) (terc (inj₂ y)) = terc (inj₂ y)

```

```

addTimeFmapLemma∞R : {lu : LUniv}
  {c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process∞ ∞ {lu} c₀)
  (a : ChoiceSet c₁)
  → fmap∞ g (addTimed✓∞ a (fmap∞ f P)) ⊑∞ addTimed✓∞ a (fmap∞ g (fmap∞ f P))
addTimeFmapLemma∞R {c₀} {c₁} {c₂} f g P a l m q = addTimeFmapLemmaR f g (forcep

```

```

addTimeFmapLemmaR : {lu : LUniv}{c₀ c₁ c₂ : Choice}
  (f : ChoiceSet c₀ → ChoiceSet c₁)
  (g : ChoiceSet c₁ → ChoiceSet c₂)
  (P : Process ∞ {lu} c₀)
  (a : ChoiceSet c₁)
  → fmap g (addTimed✓ a (fmap f P)) ⊑ addTimed✓ (g a) (fmap g (fmap f P))
addTimeFmapLemmaR f g (terminate x) a .[] .nothing (tnode empty) = tnode empty
addTimeFmapLemmaR f g (terminate x) a .(Lab (2-✓+ (g a) (g (f x))) - :: l₁) m (tnode (ext
addTimeFmapLemmaR f g (terminate x) a l m (tnode (intc .l .m () tr))
addTimeFmapLemmaR f g (terminate x) a .[] .(just (g a)) (tnode (terc zero)) = tnode (terc z
addTimeFmapLemmaR f g (terminate x) a .[] .(just (g (f x))) (tnode (terc (suc zero))) = tno
addTimeFmapLemmaR f g (terminate x) a .[] .(just (unifyA⊕A (PT (2-✓+ (g a) (g (f x))) (su
addTimeFmapLemmaR f g (node x) a l m (tnode tr) = tnode (addTimeFmapLemma+R f g x

```

A.55 libBool.agda

```
module libBool where
```

```
open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import auxData
```

```

∧introBool : (a b : Bool) → T' a → T' b → T' (a ∧ b)
∧introBool false _ ()
∧introBool true false _ ()
∧introBool true true _ _ = _

```

```

∧elimBool1 : (a b : Bool) → T' (a ∧ b) → T' a

```

```

 $\wedge$ elimBool1 false b ()
 $\wedge$ elimBool1 true b ab = _

 $\wedge$ elimBool2 : (a b : Bool)  $\rightarrow$  T' (a  $\wedge$  b)  $\rightarrow$  T' b
 $\wedge$ elimBool2 false false ()
 $\wedge$ elimBool2 true false ()
 $\wedge$ elimBool2 a true ab = _

 $\wedge$ introBool3 : (a b c : Bool)  $\rightarrow$  T' a  $\rightarrow$  T' b  $\rightarrow$  T' c  $\rightarrow$  T' (a  $\wedge$  b  $\wedge$  c)
 $\wedge$ introBool3 false _ _ ()
 $\wedge$ introBool3 true false _ _ ()
 $\wedge$ introBool3 true true false _ _ ()
 $\wedge$ introBool3 true true true _ _ _ = _

 $\wedge$ elimBool3-1 : (a b c : Bool)  $\rightarrow$  T' (a  $\wedge$  b  $\wedge$  c)  $\rightarrow$  T' a
 $\wedge$ elimBool3-1 false b c ()
 $\wedge$ elimBool3-1 true b c abc = _

 $\wedge$ elimBool3-2 : (a b c : Bool)  $\rightarrow$  T' (a  $\wedge$  b  $\wedge$  c)  $\rightarrow$  T' b
 $\wedge$ elimBool3-2 false false c ()
 $\wedge$ elimBool3-2 true false c ()
 $\wedge$ elimBool3-2 a true c abc = _

 $\wedge$ elimBool3-3 : (a b c : Bool)  $\rightarrow$  T' (a  $\wedge$  b  $\wedge$  c)  $\rightarrow$  T' c
 $\wedge$ elimBool3-3 false false false ()
 $\wedge$ elimBool3-3 true false false ()
 $\wedge$ elimBool3-3 false true false ()
 $\wedge$ elimBool3-3 true true false ()
 $\wedge$ elimBool3-3 a b true abc = _

```

A.56 libEq.agda

```

module libEq where

open import Data.Bool
open import Data.Bool.Base renaming (T to T')
open import auxData
open import libBool

```

```

==Pair : {A B : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
        (ab ab' : A × B) → Bool
==Pair _==A_ _==B_ (a , b) (a' , b') = (a ==A a') ∧ (b ==B b')

```

```

reflPair : {A B : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
          (reflA : (a : A) → T' (a ==A a))
          (reflB : (b : B) → T' (b ==B b))
          (ab : A × B)
          → T' (==Pair _==A_ _==B_ ab ab)
reflPair _==A_ _==B_ reflA reflB (a , b) = ∧introBool (a ==A a) (b ==B b) (reflA a) (reflB b)

```

```

symPair : {A B : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
          (symA : (a a' : A) (aa' : T' (a ==A a')) → T' (a' ==A a))
          (symB : (b b' : B) (bb' : T' (b ==B b')) → T' (b' ==B b))
          (ab ab' : A × B)
          (abab' : T' (==Pair _==A_ _==B_ ab ab'))
          → T' (==Pair _==A_ _==B_ ab' ab)
symPair _==A_ _==B_ symA symB (a , b) (a' , b') abab' =
  ∧introBool (a' ==A a) (b' ==B b)
    (symA a a' (∧elimBool1 (a ==A a') (b ==B b') abab'))
    (symB b b' (∧elimBool2 (a ==A a') (b ==B b') abab'))

```

```

transfPair : {A B : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
            (transfA : (a a' : A) (Q : A → Set)(aa' : T' (a ==A a')) → Q a → Q a')
            (transfB : (b b' : B) (Q : B → Set)(bb' : T' (b ==B b')) → Q b → Q b')
            (ab ab' : A × B)
            (Q : A × B → Set)
            (abab' : T' (==Pair _==A_ _==B_ ab ab'))
            (q : Q ab)
            → Q ab'
transfPair {A} {B} _==A_ _==B_ transfA transfB (a , b) (a' , b') Q abab' q
  = transfA a a'
    (λ a'' →
      (b1 b2 : B) (bb' : T' (b1 ==B b2)) →
        Q (a , b1) → Q (a'' , b2))
    (∧elimBool1 (a ==A a') (b ==B b') abab')
    (λ b b' → transfB b b' (λ b'' → Q (a , b'')) ) b b'
    (∧elimBool2 (a ==A a') (b ==B b') abab') q

```

```

==Triple : {A B C : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
           (_==C_ : C → C → Bool)
           (ab ab' : A × B × C) → Bool
==Triple _==A_ _==B_ _==C_ =
  ==Pair _==A_ (==Pair _==B_ _==C_)

```

```

reflTriple : {A B C : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
             (_==C_ : C → C → Bool)
             (reflA : (a : A) → T' (a ==A a))
             (reflB : (b : B) → T' (b ==B b))
             (reflC : (c : C) → T' (c ==C c))
             (abc : A × B × C)
             → T' (==Triple _==A_ _==B_ _==C_ abc abc)
reflTriple _==A_ _==B_ _==C_ reflA reflB reflC =
  reflPair _==A_ (==Pair _==B_ _==C_) reflA (reflPair _==B_ _==C_ reflB reflC)

```

```

symTriple : {A B C : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
            (_==C_ : C → C → Bool)
            (symA : (a a' : A) (aa' : T' (a ==A a')) → T' (a' ==A a))
            (symB : (b b' : B) (bb' : T' (b ==B b')) → T' (b' ==B b))
            (symC : (c c' : C) (cc' : T' (c ==C c')) → T' (c' ==C c))
            (abc abc' : A × B × C)
            (abcabc' : T' (==Triple _==A_ _==B_ _==C_ abc abc'))
            → T' (==Triple _==A_ _==B_ _==C_ abc' abc)
symTriple _==A_ _==B_ _==C_ symA symB symC =
  symPair _==A_ (==Pair _==B_ _==C_) symA (symPair _==B_ _==C_ symB symC)

```

```

transfTriple : {A B C : Set}(_==A_ : A → A → Bool) (_==B_ : B → B → Bool)
               (_==C_ : C → C → Bool)
               (transfA : (a a' : A) (Q : A → Set)(aa' : T' (a ==A a')) → Q a → Q a')
               (transfB : (b b' : B) (Q : B → Set)(bb' : T' (b ==B b')) → Q b → Q b')
               (transfC : (c c' : C) (Q : C → Set)(cc' : T' (c ==C c')) → Q c → Q c')
               (abc abc' : A × B × C)
               (Q : A × B × C → Set)
               (abcabc' : T' (==Triple _==A_ _==B_ _==C_ abc abc'))
               (q : Q abc)
               → Q abc'
transfTriple _==A_ _==B_ _==C_ transfA transfB transfC =
  transfPair _==A_ (==Pair _==B_ _==C_) transfA

```

```
(transfPair _==B_ _==C_ transfB transfC)
```

A.57 libList.agda

```
module libList where
```

```
open import Data.List
```

```
open import Data.List.Base renaming (map to mapL)
```

```
LUnion : {A B : Set} (lA : List A) (f : A → List B) → List B
```

```
LUnion {A} {B} lA f = concat (mapL f lA)
```

A.58 maybe.agda

```
module maybe where
```

```
data Maybe (A : Set) : Set where
```

```
  nothing : Maybe A
```

```
  just : A → Maybe A
```

A.59 monadicbind.agda

```
--@PREFIX@monadic
```

```
module monadicbind where
```

```
open import Size
```

```
open import Data.Sum
```

```
open import Data.String renaming (_++_ to _++s_)
```

```
open import Data.List.Base renaming (map to mapL)
```

```
open import choiceSetU
```

```
open import process
```

```
open import showFunction
```

```
open import dataAuxFunction
```

```
open import labelUniv
```

```
--@BEGIN@monadicBindDef
```

```
_>>=Str_ : {c0 : Choice} → String
           → (ChoiceSet c0 → String) → String
s >>=Str f = s ++s ">>" ++s choice2Str2Str f
```

```
mutual
```

```
_>>=∞_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
        → Process∞ i {lu} c0
        → (ChoiceSet c0 → Process∞ i {lu} c1)
        → Process∞ i {lu} c1
forcep (P >>=∞ Q) = forcep P >>= Q
Str∞ (P >>=∞ Q) = Str∞ P >>=Str (Str∞ ∘ Q)
```

```
_>>=_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
        → Process i c0
        → (ChoiceSet c0 → Process∞ (↑ i) {lu} c1)
        → Process i c1
node P >>= Q = node (P >>=+ Q)
terminate x >>= Q = forcep (Q x)
```

```
_>>=+_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
        → Process+ i c0
        → (ChoiceSet c0 → Process∞ i {lu} c1)
        → Process+ i c1
E (P >>=+ Q) = E P
Lab (P >>=+ Q) = Lab P
PE (P >>=+ Q) c = PE P c >>=∞ Q
I (P >>=+ Q) = I P Ψ' T P
PI (P >>=+ Q) (inj1 c) = PI P c >>=∞ Q
PI (P >>=+ Q) (inj2 c) = Q (PT P c)
T (P >>=+ Q) = ∅'
PT (P >>=+ Q) ()
Str+ (P >>=+ Q) = Str+ P >>=Str (Str∞ ∘ Q)
```

```
--@END
```

```
_>>=+p_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process+ i c0
        → (ChoiceSet c0 → Process∞ i {lu} c1)
```

$$P \gg_{+p} Q = \text{node} (P \gg_{+} Q) \rightarrow \text{Process } i \ c_1$$

A.60 monadicbindFixing2ndMonadLaw.agda

```
--@PREFIX@monadicFixingSecondMonadLaw
```

```
module monadicbindFixing2ndMonadLaw where
```

```
open import Size
open import Data.Sum
open import Data.String renaming (_++_ to _++s_)
open import Data.List.Base renaming (map to mapL)
open import choiceSetU
open import process
open import showFunction
open import dataAuxFunction
open import labelUniv
open import Data.Bool renaming (T to True)
open import auxData
```

```
--@BEGIN@isTerminate
```

$$\begin{aligned} \text{isTerminate} &: \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process } i \{lu\} c) \\ &\quad \rightarrow \text{Bool} \\ \text{isTerminate } (\text{terminate } x) &= \text{true} \\ \text{isTerminate } (\text{node } x) &= \text{false} \end{aligned}$$
$$\begin{aligned} \text{isNode} &: \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process } i \{lu\} c) \\ &\quad \rightarrow \text{Bool} \\ \text{isNode} &= \neg b \circ \text{isTerminate} \end{aligned}$$

--@END

```
--@BEGIN@processIsTerminateToResult
```

$$\text{processIsTerminateToResult} : \{i : \text{Size}\} \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P : \text{Process } i \{lu\} c)$$

```

      (isTer : True (isTerminate P))
      → ChoiceSet c
processIsTerminateToResult (terminate x) isTer = x
processIsTerminateToResult (node x) ()

--@END

processNotTerminate2Process+ : {i : Size}{lu : LUniv} {c : Choice}
      (P : Process i {lu} c)
      (notTer : True (isNode P))
      → Process+ i {lu} c
processNotTerminate2Process+ (terminate x) ()
processNotTerminate2Process+ (node P) notTer = P

--@BEGIN@monadicBindDef

_>>=Str_ : {c0 : Choice} → String
      → (ChoiceSet c0 → String) → String
s >>=Str f = s ++s ">>" ++s choice2Str2Str f

mutual
  _>>=∞_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
      → Process∞ i {lu} c0
      → (ChoiceSet c0 → Process i {lu} c1)
      → Process∞ i {lu} c1
  forcep (P >>=∞ Q) = forcep P >>= Q
  Str∞ (P >>=∞ Q) = Str∞ P >>=Str (Str ∘ Q)

  _>>=_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
      → Process i c0
      → (ChoiceSet c0 → Process i {lu} c1)
      → Process i c1
  node P >>= Q = node (P >>=+ Q)
  terminate x >>= Q = Q x

  _>>=+_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice}
      → Process+ i c0
      → (ChoiceSet c0 → Process i {lu} c1)

```

```

      → Process+ i c1
E   (P >>=+ Q)      = E   P
Lab (P >>=+ Q)      = Lab P
PE  (P >>=+ Q) c    = PE  P c >>=∞ Q
I   (P >>=+ Q)      = I P ⊔' subset' (T P)
                                   (isNode ∘ (Q ∘ (PT P)))
PI  (P >>=+ Q) (inj1 c) = PI P c >>=∞ Q
forcep (PI (P >>=+ Q) (inj2 (sub a x))) = Q (PT P a)
Str∞ (PI (P >>=+ Q) (inj2 (sub a x))) = Str (Q (PT P a))
T   (P >>=+ Q)      = subset' (T P)
                                   (isTerminate ∘ (Q ∘ (PT P)))
PT  (P >>=+ Q) (sub a x) = processIsTerminateToResult (Q (PT P a)) x
Str+ (P >>=+ Q)      = Str+ P >>=Str (Str ∘ Q)

--@END

_>>=+p_ : {i : Size} → {lu : LUniv} → {c0 c1 : Choice} → Process+ i c0
      → ( ChoiceSet c0 → Process i {lu} c1)
      → Process i c1
P >>=+p Q = node (P >>=+ Q)

```

A.61 NativeIO.agda

```
module NativeIO where
```

```
open import UnitModule
open import Data.String.Base using (String)
```

```
postulate
```

```
NativeIO    : Set → Set
```

```
nativeReturn : {A : Set} → A → NativeIO A
```

```
_native»=_ : {A B : Set} → NativeIO A → (A → NativeIO B) → NativeIO B
```

```
-- {-# FOREIGN GHC import qualified IO.FFI #-}
```

```
{-# BUILTIN IO NativeIO #-}
```

```
--{-# COMPILED_TYPE NativeIO IO #-} -- IO.FFI.AgdaIO
```

```
{-# COMPILE GHC NativeIO = type IO #-}
```

```
-- MAlonzo.Code.Agda.Builtin.IO.AgdaIO
```

```
{-# COMPILER GHC _native»=_ = \_ _ -> (»=) :: IO a -> (a -> IO b) -> IO b #-}
{-# COMPILER GHC nativeReturn = \_ -> return :: a -> IO a #-}
```

```
postulate
```

```
  nativeGetLine  : NativeIO String
  nativePutStrLn : String → NativeIO Unit
```

```
{-# COMPILER GHC nativePutStrLn = \ s -> putStrLn (Data.Text.unpack s) #-}
{-# COMPILER GHC nativeGetLine = fmap Data.Text.pack getLine #-}
```

A.62 parallelSimple.agda

```
--@PREFIX@parallel
```

```
module parallelSimple where
```

```
-- version of parallel which is as in standard CSP
```

```
open import Data.Bool renaming (T to True)
open import Data.Sum
open import Data.String renaming (_++_ to _++s_)
open import Size
open import process
open import labelUniv
open import auxData
open import dataAuxFunction
open import choiceSetU
open import renamingResult
open import restrict
```

```
_ \_ : {X : Set} → (A B : X → Bool) → X → Bool
(A \ B) c = A c ∧ (¬b (B c))
```

```
-- \ is input as \setminus
```

```
--@BEGIN@parallelinfDef
```

mutual

--@END

```
--@BEGIN@parallelDef
```

--@END

$$\begin{aligned} \text{--}[\text{--}]||\infty+\text{--}[\text{--}]\text{--} &: \{i : \text{Size}\} \{lu : \text{LUniv}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ &\rightarrow \text{Process}\infty \ i \ \{lu\} \ c_0 \\ &\rightarrow (A \ B : \text{Label} \ lu \rightarrow \text{Bool}) \\ &\rightarrow \text{Process+} \ i \ \{lu\} \ c_1 \end{aligned}$$

$$\begin{aligned}
& \rightarrow \text{Process}\infty i (c_0 \times' c_1) \\
\text{forcep } (P [A] || \infty + [B] Q) &= \text{node } (\text{forcep } P [A] || \text{p} + [B] Q) \\
\text{Str}\infty (P [A] || \infty + [B] Q) &= \text{Str}\infty P [A] || \text{Str} [B] \text{Str} + Q
\end{aligned}$$

$$\begin{aligned}
-[-] || \infty + [-] - : \{lu : \text{LUniv}\} \{i : \text{Size}\} &\rightarrow \{c_0 c_1 : \text{Choice}\} \\
&\rightarrow \text{Process} + i \{lu\} c_0 \\
&\rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) \\
&\rightarrow \text{Process}\infty i \{lu\} c_1 \\
&\rightarrow \text{Process}\infty i \{lu\} (c_0 \times' c_1) \\
\text{forcep } (P [A] || \infty + [B] Q) &= \text{node } (P [A] || \text{p} + [B] \text{forcep } Q) \\
\text{Str}\infty (P [A] || \infty + [B] Q) &= \text{Str} + P [A] || \text{Str} [B] \text{Str}\infty Q
\end{aligned}$$

$$\begin{aligned}
-[-] || \text{p} + [-] - : \{lu : \text{LUniv}\} \{i : \text{Size}\} &\rightarrow \{c_0 c_1 : \text{Choice}\} \\
&\rightarrow \text{Process } i \{lu\} c_0 \\
&\rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) \\
&\rightarrow \text{Process} + i \{lu\} c_1 \\
&\rightarrow \text{Process} + i \{lu\} (c_0 \times' c_1) \\
(\text{terminate } a) [A] || \text{p} + [B] Q &= \text{fmap} + (\lambda b \rightarrow (a \text{ ,, } b))(Q \uparrow + (B \setminus A)) \\
(\text{node } P) [A] || \text{p} + [B] Q &= P [A] || \text{p} + [B] Q
\end{aligned}$$

$$\begin{aligned}
-[-] || \text{p} [-] - : \{i : \text{Size}\} \{lu : \text{LUniv}\} &\rightarrow \{c_0 c_1 : \text{Choice}\} \\
&\rightarrow \text{Process} + i \{lu\} c_0 \\
&\rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) \\
&\rightarrow \text{Process } i \{lu\} c_1 \\
&\rightarrow \text{Process} + i \{lu\} (c_0 \times' c_1) \\
P [A] || \text{p} + [B] \text{terminate } b &= \text{fmap} + (\lambda a \rightarrow (a \text{ ,, } b))(P \uparrow + (A \setminus B)) \\
P [A] || \text{p} + [B] \text{node } Q &= P [A] || \text{p} + [B] Q
\end{aligned}$$

--@BEGIN@parallelplusDef

$$\begin{aligned}
-[-] || + [-] - : \{lu : \text{LUniv}\} \{i : \text{Size}\} &\rightarrow \{c_0 c_1 : \text{Choice}\} \\
&\rightarrow \text{Process} + i \{lu\} c_0 \\
&\rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) \\
&\rightarrow \text{Process} + i \{lu\} c_1 \\
&\rightarrow \text{Process} + i \{lu\} (c_0 \times' c_1) \\
\text{E } (P [A] || + [B] Q) &= \text{subset}' (\text{E } P) ((A \setminus B) \circ (\text{Lab } P)) \uplus'
\end{aligned}$$

```

subset' (E Q) ((B \ A) ∘ (Lab Q)) ⊔'
subset' (E P ×' E Q)
      (λ { (e₁ ,, e₂)
→ Lab P e₁ == Lab Q e₂ ∧ A (Lab P e₁) ∧ B (Lab Q e₂) })
Lab (P [ A ]||+[ B ] Q) (inj₁ (inj₁ (sub c p))) = Lab P c
Lab (P [ A ]||+[ B ] Q) (inj₁ (inj₂ (sub c p))) = Lab Q c
Lab (P [ A ]||+[ B ] Q) (inj₂ (sub (c₀ ,, c₁) p)) = Lab P c₀
PE (P [ A ]||+[ B ] Q) (inj₁ (inj₁ (sub c p))) = PE P c [ A ]||∞+[ B ] Q
PE (P [ A ]||+[ B ] Q) (inj₁ (inj₂ (sub c p))) = P [ A ]||+∞[ B ] PE Q c
PE (P [ A ]||+[ B ] Q) (inj₂ (sub (c₀ ,, c₁) p)) = PE P c₀ [ A ]||∞[ B ] PE Q c₁
I (P [ A ]||+[ B ] Q) = I P ⊔' I Q
PI (P [ A ]||+[ B ] Q) (inj₁ c) = PI P c [ A ]||∞+[ B ] Q
PI (P [ A ]||+[ B ] Q) (inj₂ c) = P [ A ]||+∞[ B ] PI Q c
T (P [ A ]||+[ B ] Q) = T P ×' T Q
PT (P [ A ]||+[ B ] Q) (c₀ ,, c₁) = (PT P c₀ ,, PT Q c₁)
Str+ (P [ A ]||+[ B ] Q) = Str+ P [ A ]||Str[ B ] Str+ Q

```

--@END

mutual

```

_[-]||wNam∞[-]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process∞ i {lu} c₀
→ (A B : Label lu → Bool)
→ Process∞ i {lu} c₁
→ (///name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process∞ i {lu} (c₀ ×' c₁)
forcep ( P [ A ]||wNam∞[ B ] Q Using ///name , fmapLeftName , fmapRightName )
= forcep P [ A ]||wNam[ B ] forcep Q Using ///name , fmapLeftName , fmap
Str∞ ( P [ A ]||wNam∞[ B ] Q Using ///name , fmapLeftName , fmapRightName )
= ///name (Str∞ P) (Str∞ Q)

_[-]||wNam[-]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process i {lu} c₀
→ (A B : Label lu → Bool)
→ Process i {lu} c₁
→ (///name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)

```

```

→ Process i {lu} (c₀ ×' c₁)
node P [ A ]||wNam[ B ] node Q Using [///]name , fmapLeftName , fmapRightName
= node (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
terminate a [ A ]||wNam[ B ] Q Using [///]name , fmapLeftName , fmapRightName
= fmapWithName (fmapLeftName a) (λ b → (a „ b))(Q ↑ (B \ A))
P [ A ]||wNam[ B ] terminate b Using [///]name , fmapLeftName , fmapRightName
= fmapWithName (fmapRightName b) (λ a → (a „ b))(P ↑ (A \ B))

```

```

-[_]||wNam∞+[_]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process∞ i {lu} c₀
→ (A B : Label lu → Bool)
→ Process+ i {lu} c₁
→ ([///]name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process∞ i {lu} (c₀ ×' c₁)
forcep (P [ A ]||wNam∞+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
= node (forcep P [ A ]||wNam+p[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
Str∞ (P [ A ]||wNam∞+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
= [///]name (Str∞ P) (Str+ Q)

```

```

-[_]||wNam+∞[_]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process+ i {lu} c₀
→ (A B : Label lu → Bool)
→ Process∞ i {lu} c₁
→ ([///]name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process∞ i {lu} (c₀ ×' c₁)
forcep (P [ A ]||wNam+∞[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
= node (P [ A ]||wNam+p[ B ] forcep Q Using [///]name , fmapLeftName , fmapRightName)
Str∞ (P [ A ]||wNam+∞[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
= [///]name (Str+ P) (Str∞ Q)

```

```

-[_]||wNamp+[_]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process i {lu} c₀
→ (A B : Label lu → Bool)
→ Process+ i {lu} c₁

```

```

→ (///name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process+ i {lu} (c₀ ×' c₁)
(terminate a) [ A ]||wNamp+[ B ] Q Using ///name , fmapLeftName , fmapRightName
= fmapWithName+ (fmapLeftName a) (λ b → (a ,, b))(Q ↑+ (B \ A) )
(node P) [ A ]||wNamp+[ B ] Q Using ///name , fmapLeftName , fmapRightName
= P [ A ]||wNam+[ B ] Q Using ///name , fmapLeftName , fmapRightName

```

```

-[ ]||wNam+p[ ]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process+ i {lu} c₀
→ (A B : Label lu → Bool)
→ Process i {lu} c₁
→ (///name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process+ i {lu} (c₀ ×' c₁)
P [ A ]||wNam+p[ B ] terminate b Using ///name , fmapLeftName , fmapRightName
= fmapWithName+ (fmapRightName b) (λ a → (a ,, b))(P ↑+ (A \ B))
P [ A ]||wNam+p[ B ] node Q Using ///name , fmapLeftName , fmapRightName
= P [ A ]||wNam+[ B ] Q Using ///name , fmapLeftName , fmapRightName

```

```

-[ ]||wNam+[ ]_Using_,_,_ : {i : Size} → {c₀ c₁ : Choice} {lu : LUniv}
→ Process+ i {lu} c₀
→ (A B : Label lu → Bool)
→ Process+ i {lu} c₁
→ (///name : String → String → String)
→ (fmapLeftName : ChoiceSet c₀ → String → String)
→ (fmapRightName : ChoiceSet c₁ → String → String)
→ Process+ i {lu} (c₀ ×' c₁)
E (P [ A ]||wNam+[ B ] Q Using ///name , fmapLeftName , fmapRightName)
= subset' (E P) ((¬b ○ A) ○ (Lab P)) ⊕'
    subset' (E Q) ((¬b ○ B) ○ (Lab Q)) ⊕'
    subset' (E P ×' E Q) (λ {(e₁ ,, e₂)}
      → Lab P e₁ ==| Lab Q e₂ ∧ A (Lab P e₁) ∧ B (Lab Q e₂))
Lab (P [ A ]||wNam+[ B ] Q Using ///name , fmapLeftName , fmapRightName) (inj₁ (inj₁
  = Lab P c
Lab (P [ A ]||wNam+[ B ] Q Using ///name , fmapLeftName , fmapRightName) (inj₁ (inj₂

```

```

    = Lab Q c
Lab (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj2 (sub (c0 ,, c1))
    = Lab P c0
PE (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj1 (inj1 (sub c p
    = PE P c [ A ]||wNam∞+[ B ] Q Using [///]name , fmapLeftName , fmapRightName
PE (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj1 (inj2 (sub c p
    = P [ A ]||wNam+∞[ B ] PE Q c Using [///]name , fmapLeftName , fmapRightName
PE (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj2 (sub (c0 ,, c1))
    = PE P c0 [ A ]||wNam∞[ B ] PE Q c1 Using [///]name , fmapLeftName , fmapRightName
I (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
    = I P ⊕' I Q
PI (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj1 c)
    = PI P c [ A ]||wNam∞+[ B ] Q Using [///]name , fmapLeftName , fmapRightName
PI (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (inj2 c)
    = P [ A ]||wNam+∞[ B ] PI Q c Using [///]name , fmapLeftName , fmapRightName
T (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
    = T P ×' T Q
PT (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName) (c0 ,, c1)
    = PT P c0 ,, PT Q c1
Str+ (P [ A ]||wNam+[ B ] Q Using [///]name , fmapLeftName , fmapRightName)
    = [///]name (Str+ P) (Str+ Q)

```

A.63 prefix.agda

```

--@PREFIX@prefix

module prefix where

open import Size
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import process
open import choiceSetU
open import showFunction
open import dataAuxFunction
open import labelUniv
-- open import NativeIO

```



--@BEGIN@preDef

$\longrightarrow_{\text{Str}} : \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{String} \rightarrow \text{String}$
 $l \longrightarrow_{\text{Str}} s = "(" ++ \text{showLabel } l ++ s " \rightarrow " ++ s ++ s "$

$\longrightarrow_{+} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu$
 $\rightarrow \text{Process}_{\infty} i c \rightarrow \text{Process}_{+} i c$

E $(l \longrightarrow_{+} P) = \top'$
Lab $(l \longrightarrow_{+} P) c = l$
PE $(l \longrightarrow_{+} P) c = P$
I $(l \longrightarrow_{+} P) = \emptyset'$
PI $(l \longrightarrow_{+} P) ()$
T $(l \longrightarrow_{+} P) = \emptyset'$
PT $(l \longrightarrow_{+} P) ()$
Str+ $(l \longrightarrow_{+} P) = l \longrightarrow_{\text{Str}} \text{Str}_{\infty} P$

$\longrightarrow_{-} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu$
 $\rightarrow \text{Process}_{\infty} i c \rightarrow \text{Process } i c$
 $l \longrightarrow P = \text{node } (l \longrightarrow_{+} P)$

--@END

$\longrightarrow_{+'} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process}_{\infty} i c \rightarrow \text{Process}_{+'} i c$
 $l \longrightarrow_{+'} P = \text{process}_{+} \top' (\lambda _ \rightarrow l) (\lambda _ \rightarrow P) \emptyset' \text{efq } \emptyset' \text{efq}$
 $(l \longrightarrow_{\text{Str}} \text{Str}_{\infty} P)$

$\longrightarrow_{++} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process}_{+} i c \rightarrow \text{Process}_{++} i c$
 $l \longrightarrow_{++} P = l \longrightarrow_{+} (\text{delay } (\text{node } P))$

$\longrightarrow_{p+} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process } i c \rightarrow \text{Process}_{+} i c$
 $l \longrightarrow_{p+} P = l \longrightarrow_{+} (\text{delay } P)$

$\longrightarrow_{pp} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process } i c \rightarrow \text{Process } i c$
 $l \longrightarrow_{pp} P = \text{node } (l \longrightarrow_{+} (\text{delay } P))$

$\longrightarrow_{p\infty} : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process } i c \rightarrow \text{Process}_{\infty} i c$
forcep $(l \longrightarrow_{p\infty} P) = l \longrightarrow_{pp} P$
Str ∞ $(l \longrightarrow_{p\infty} P) = l \longrightarrow_{\text{Str}} \text{Str } P$



```

 $\_ \longrightarrow \infty \_ : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process} \infty i \{lu\} c \rightarrow \text{Process} \infty i \{lu\} c$ 
 $\text{forcep } (l \longrightarrow \infty P) = l \longrightarrow P$ 
 $\text{Str} \infty (l \longrightarrow \infty P) = l \longrightarrow \text{Str } \text{Str} \infty P$ 

```

A.64 preFix.agda

```
module preFix where
```

```

open import Size
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import process
open import choiceSetU
open import showFunction
open import dataAuxFunction
open import labelUniv
-- open import NativeIO

```

```

 $\_ \longrightarrow \text{Str} \_ : \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{String} \rightarrow \text{String}$ 
 $l \longrightarrow \text{Str } s = \text{"(" ++s showLabel l ++s " } \longrightarrow \text{" ++s s ++s "}"}$ 
 $\text{"(" ++s showLabel l ++s " } -> \text{" ++s s ++s "}"}$ 

```

```

 $\_ \longrightarrow + \_ : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process} \infty i c \rightarrow \text{Process} + i c$ 
 $l \longrightarrow + P = \text{process} + \top' (\lambda \_ \rightarrow l) (\lambda \_ \rightarrow P) \emptyset' \text{efq } \emptyset' \text{efq}$ 
 $(l \longrightarrow \text{Str } \text{Str} \infty P)$ 

```

```

 $\_ \longrightarrow \_ : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process} \infty i c \rightarrow \text{Process } i c$ 
 $l \longrightarrow P = \text{node } (l \longrightarrow + P)$ 

```

```

{- Nicer looking version, use in library instead of  $\_ \longrightarrow + \_$  but not in paper -}
 $\_ \longrightarrow +' \_ : \{i : \text{Size}\} \rightarrow \{c : \text{Choice}\} \rightarrow \{lu : \text{LUniv}\} \rightarrow \text{Label } lu \rightarrow \text{Process} \infty i c \rightarrow \text{Process} + i c$ 
E (l  $\longrightarrow +'$  P) =  $\top'$ 
Lab (l  $\longrightarrow +'$  P) c = l
PE (l  $\longrightarrow +'$  P) c = P
I (l  $\longrightarrow +'$  P) =  $\emptyset'$ 
PI (l  $\longrightarrow +'$  P) ()
T (l  $\longrightarrow +'$  P) =  $\emptyset'$ 

```

```

PT (l →+' P) ()
Str+ (l →+' P) = showLabel l -- 1 →Str Str∞ P

_→++_ : {i : Size} → {c : Choice} → {lu : LUniv} → Label lu → Process+ i c → Process+
l →++ P = l →+ (delay (node P) )

_→p+_ : {i : Size} → {c : Choice} → {lu : LUniv} → Label lu → Process i c → Process+
l →p+ P = l →+ (delay P)

_→pp_ : {i : Size} → {c : Choice} → {lu : LUniv} → Label lu → Process i c → Process (↑
l →pp P = node (l →+ (delay P) )

_→p∞_ : {i : Size} → {c : Choice} → {lu : LUniv} → Label lu → Process i c → Process∞
forcep (l →p∞ P) = l →pp P
Str∞ (l →p∞ P) = l →Str Str P

_→∞∞_ : {i : Size} → {c : Choice} → {lu : LUniv} → Label lu → Process∞ i {lu} c → Process∞
forcep (l →∞∞ P) = l → P
Str∞ (l →∞∞ P) = l →Str Str∞ P

```

A.65 primitiveProcess.agda

```
--@PREFIX@primitive
```

```
module primitiveProcess where
```

```

open import Size
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List
open import process
open import auxData
open import dataAuxFunction
open import choiceSetU
open import labelUniv

```

```
--@BEGIN@StopDef
```

```

STOP+ : {i : Size} → (c : Choice) → {lu : LUniv} → Process+ i {lu} c
STOP+ c = process+ ∅' efq ∅' efq ∅' efq "STOP"

```

```

STOP : {i : Size} → (c : Choice) → {lu : LUniv} → Process i {lu} c
STOP c = node (STOP+ c)

STOP∞ : {i : Size} → (c : Choice) → {lu : LUniv} → Process∞ i {lu} c
forcep (STOP∞ c) = STOP c
Str∞ (STOP∞ c) = "STOP∞"

--@END

MSKIP+ : (i : Size) → (c : Choice) → (t : Choice) → {lu : LUniv}
        → (f : ChoiceSet t → ChoiceSet c) → Process+ i {lu} c
MSKIP+ i c t f = process+ ∅' efq efq ∅' efq t f "MSKIP"

MSKIP : (i : Size) → (c : Choice) → (t : Choice) → {lu : LUniv}
        → (f : ChoiceSet t → ChoiceSet c) → Process i {lu} c
MSKIP i c t f = node (MSKIP+ i c t f)

MSkip : {i : Size} → {c : Choice} → (t : Choice) → {lu : LUniv}
        → (f : ChoiceSet t → ChoiceSet c) → Process i {lu} c
MSkip {i} {c} = MSKIP i c

--@BEGIN@SkipDef

SKIP+ : {i : Size} → {c : Choice} → (a : ChoiceSet c)
        → {lu : LUniv} → Process+ i {lu} c
SKIP+ a = process+ ∅' efq efq ∅' efq T' (λ _ → a)
        ("SKIP(" ++s choice2Str a ++s ")")

SKIP : {i : Size} → {c : Choice} → (a : ChoiceSet c)
        → {lu : LUniv} → Process i {lu} c
SKIP a = node (SKIP+ a)

SKIP∞ : {i : Size} → {c : Choice} → (a : ChoiceSet c)
        → {lu : LUniv} → Process∞ i {lu} c
forcep (SKIP∞ a) = SKIP a
Str∞ (SKIP∞ a) = ("SKIP∞(" ++s choice2Str a ++s ")")

```

```
--@END
```

```
TERMINATE : {i : Size} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv} → Process i {lu}
TERMINATE a = terminate a
```

```
TERMINATE∞ : {i : Size} → {c : Choice} → (a : ChoiceSet c) → {lu : LUniv} → Process∞ i {lu}
forcep (TERMINATE∞ a) = TERMINATE a
Str∞ (TERMINATE∞ a) = "terminate(" ++s choice2Str a ++s ")"
```

```
SKIPL+ : {i : Size} → {c : Choice} → List (ChoiceSet c) → {lu : LUniv} → Process+ i {lu}
SKIPL+ {i} {c} l = process+ ∅' efq efq ∅' efq (fin (length l)) (nth l) "SKIPL ???"
```

A.66 process.agda

```
--@PREFIX@Process
```

```
module process where
```

```
open import choiceSetU
open import labelUniv
open import Size
open import Data.String renaming (_++_ to _++s_)
```

```
--@BEGIN@processinf
```

```
mutual
  record Process∞ (i : Size) {lu : LUniv} (c : Choice) : Set where
    coinductive
    field
      forcep : {j : Size< i} → Process j {lu} c
      Str∞ : String
```

```
--@END
```

```
--@BEGIN@process
```

```
data Process (i : Size) {lu : LUniv} (c : Choice) : Set where
```

```

    terminate : ChoiceSet c → Process i {lu} c
    node      : Process+ i {lu} c → Process i {lu} c

--@END

--@BEGIN@processplus

record Process+ (i : Size) {lu : LUniv} (c : Choice) : Set where
  constructor process+
  coinductive
  field
    E      : Choice
    Lab    : ChoiceSet E → Label lu
    PE     : ChoiceSet E → Process∞ i {lu} c
    I      : Choice
    PI     : ChoiceSet I → Process∞ i {lu} c
    T      : Choice
    PT     : ChoiceSet T → ChoiceSet c
    Str+   : String
--@END

open Process∞ public
open Process+ public

Ep : {i : Size}{lu : LUniv} {c : Choice} → Process i {lu} c → Choice
Ep {i} {c} (terminate x) = ∅'
Ep {i} {c} (node Q) = E Q

Labp : {i : Size}{lu : LUniv}{c : Choice} → (P : Process i {lu} c) → ChoiceSet (Ep P) → Label lu
Labp {i} {c} (terminate x) ()
Labp {i} {c} (node Q) x = Lab Q x

E∞ : {lu : LUniv}{c : Choice} → Process∞ ∞ {lu} c → Choice
E∞ {c} P = Ep (forcep P)

Lab∞ : {lu : LUniv} {c : Choice} → (P : Process∞ ∞ {lu} c) → ChoiceSet (E∞ P) → Label lu
Lab∞ {c} P x = Labp (forcep P) x

```

```

lp : {i : Size}{lu : LUniv}{c : Choice} → Process i {lu} c → Choice
lp {i} {c} (terminate x) = ∅'
lp {i} {c} (node Q) = I Q

Tp : {i : Size}{lu : LUniv}{c : Choice} → Process i {lu} c → Choice
Tp {i} {c} (terminate x) = ∅'
Tp {i} {c} (node Q) = T Q

Str : {i : Size} → {c : Choice} → {lu : LUniv} → Process i {lu} c → String
Str (terminate a) = "terminate(++s choice2Str a ++s)"
Str (node P) = Str+ P

--@BEGIN@delayprocess

delay : {i : Size} → {lu : LUniv} → {c : Choice} → Process i {lu} c
      → Process∞ (↑ i) {lu} c
forcep (delay P) = P
Str∞ (delay P) = Str P

--@END

delayi : (i : Size) → {lu : LUniv} → {c : Choice} → Process i {lu} c
      → Process∞ (↑ i) {lu} c
forcep (delayi i P) = P
Str∞ (delayi i P) = Str P

```

A.67 process2OptimizedProcess.agda

```
module process2OptimizedProcess where
```

```
open import choiceSetU
open import choiceSetUOptimized
open import labelUniv
```

```

open import Size
open import Data.String renaming (_++_ to _++s_)
open import process

mutual
  optimizedProcess∞ : (i : Size) {lu : LUniv}(c : Choice) (p : Process∞ i {lu} c)
    → Process∞ i {lu} c
  forcep (optimizedProcess∞ i c p) {j} = optimizedProcess j c (forcep p {j})
  Str∞ (optimizedProcess∞ i c p) = Str∞ p

  optimizedProcess : (i : Size) {lu : LUniv}(c : Choice) (p : Process i {lu} c)
    → Process i {lu} c
  optimizedProcess i c (terminate x) = terminate x
  optimizedProcess i c (node x) = node (optimizedProcess+ i c x)

  optimizedProcess+ : (i : Size) {lu : LUniv}(c : Choice) (p : Process+ i {lu} c)
    → Process+ i {lu} c
  E (optimizedProcess+ i c p) = choice2OptimizedChoice (E p)
  Lab (optimizedProcess+ i c p) x = Lab p (choice2OptimizedChoice2choice (E p) x)
  PE (optimizedProcess+ i c p) x = optimizedProcess∞ i c (PE p (choice2OptimizedChoice2choice (E p) x))
  I (optimizedProcess+ i c p) = choice2OptimizedChoice (I p)
  PI (optimizedProcess+ i c p) x = optimizedProcess∞ i c (PI p (choice2OptimizedChoice2choice (I p) x))
  T (optimizedProcess+ i c p) = choice2OptimizedChoice (T p)
  PT (optimizedProcess+ i c p) x = PT p (choice2OptimizedChoice2choice (T p) x)
  Str+ (optimizedProcess+ i c p) = Str+ p

```

A.68 process2OptimizedProcess2.agda

```

module process2OptimizedProcess2 where

open import choiceSetU
open import choiceSetUOptimized2
open import labelUniv
open import Size
open import Data.String renaming (_++_ to _++s_)
open import process

```

mutual

optimizedProcess ∞ : (i : Size) {lu : LUniv}(c : Choice) (p : Process ∞ i {lu} c)
 \rightarrow Process ∞ i {lu} c

forcep (optimizedProcess ∞ i c p) {j} = optimizedProcess j c (forcep p {j})

Str ∞ (optimizedProcess ∞ i c p) = Str ∞ p

optimizedProcess : (i : Size) {lu : LUniv}(c : Choice) (p : Process i {lu} c)
 \rightarrow Process i {lu} c

optimizedProcess i c (terminate x) = terminate x

optimizedProcess i c (node x) = node (optimizedProcess+ i c x)

optimizedProcess+ : (i : Size) {lu : LUniv}(c : Choice) (p : Process+ i {lu} c)
 \rightarrow Process+ i {lu} c

E (optimizedProcess+ i c p) = choice2OptimizedChoice (E p)

Lab (optimizedProcess+ i c p) x = Lab p (choice2OptimizedChoice2choice (E p) x)

PE (optimizedProcess+ i c p) x = optimizedProcess ∞ i c (PE p (choice2OptimizedChoice2choice (E p) x))

I (optimizedProcess+ i c p) = choice2OptimizedChoice (I p)

PI (optimizedProcess+ i c p) x = optimizedProcess ∞ i c (PI p (choice2OptimizedChoice2choice (I p) x))

T (optimizedProcess+ i c p) = choice2OptimizedChoice (T p)

PT (optimizedProcess+ i c p) x = PT p (choice2OptimizedChoice2choice (T p) x)

Str+ (optimizedProcess+ i c p) = Str+ p

A.69 process2OptimizedProcess3.agda

module process2OptimizedProcess3 where

open import choiceSetU

open import choiceSetUOptimized3

open import labelUniv

open import Size

open import Data.String renaming (_++_ to _++s_)

open import process

mutual

optimizedProcess ∞ : (i : Size) {lu : LUniv}(c : Choice) (p : Process ∞ i {lu} c)
 \rightarrow Process ∞ i {lu} c

```

forcep (optimizedProcess $\infty$  i c p) {j} = optimizedProcess j c (forcep p {j})
Str $\infty$  (optimizedProcess $\infty$  i c p) = Str $\infty$  p

```

```

optimizedProcess : (i : Size) {lu : LUniv}(c : Choice) (p : Process i {lu} c)
  → Process i {lu} c

```

```

optimizedProcess i c (terminate x) = terminate x

```

```

optimizedProcess i c (node x) = node (optimizedProcess+ i c x)

```

```

optimizedProcess+ : (i : Size) {lu : LUniv}(c : Choice) (p : Process+ i {lu} c)
  → Process+ i {lu} c

```

```

E (optimizedProcess+ i c p) = choice2OptimizedChoice (E p)

```

```

Lab (optimizedProcess+ i c p) x = Lab p (choice2OptimizedChoice2choice (E p) x)

```

```

PE (optimizedProcess+ i c p) x = optimizedProcess $\infty$  i c (PE p (choice2OptimizedChoice2choice (E p) x))

```

```

I (optimizedProcess+ i c p) = choice2OptimizedChoice (I p)

```

```

PI (optimizedProcess+ i c p) x = optimizedProcess $\infty$  i c (PI p (choice2OptimizedChoice2choice (I p) x))

```

```

T (optimizedProcess+ i c p) = choice2OptimizedChoice (T p)

```

```

PT (optimizedProcess+ i c p) x = PT p (choice2OptimizedChoice2choice (T p) x)

```

```

Str+ (optimizedProcess+ i c p) = Str+ p

```

A.70 proofAss.agda

```

--@PREFIX@mainproofAss

```

```

module proofAss where

```

```

open import process
open import TraceWithoutSize
open import Size
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.Product
open import Interleave
open import Data.List
open import Data.Sum
open import renamingResult
open import RefWithoutSize

```

```

open import dataAuxFunction
open import lemFmap
open import traceEquivalence
open import Data.Product
open import labelUniv
open import auxData

```

```

maybeChoice : {c0 c1 c2 : Choice} → Set
maybeChoice {c0} {c1} {c2} = ((ChoiceSet c0 auxData.× ChoiceSet c1) auxData.× ChoiceSet

```

```

mutual

```

```

--@BEGIN@AssIntDef

```

```

Ass|||+ : {lu : LUniv}{c0 c1 c2 : Choice}    (P : Process+ ∞ {lu} c0)
          (Q : Process+ ∞ {lu} c1)
          (Z : Process+ ∞ {lu} c2)
→ ((P |||++ Q) |||++ Z) ⊑+ fmap+ Ass× (P |||++ (Q |||++ Z))
Ass|||+ P Q Z .[] .nothing empty = empty
Ass|||+ P Q Z .(Lab P x :: l) m (extc l .m (inj1 x) x1)
    = extc l m (inj1 (inj1 x)) (Ass|||∞++ (PE P x) Q Z l m x1)
Ass|||+ P Q Z .(Lab Q x :: l) m (extc l .m (inj2 (inj1 x)) x1)
    = extc l m (inj1 (inj2 x)) (Ass|||+∞++ P (PE Q x) Z l m x1)
Ass|||+ P Q Z .(Lab Z y :: l) m (extc l .m (inj2 (inj2 y)) x1)
    = extc l m (inj2 y) (Ass|||+∞++ P Q (PE Z y) l m x1)
Ass|||+ P Q Z l m (intc l .m (inj1 x) x1)
    = intc l m (inj1 (inj1 x)) (Ass|||∞++ (PI P x) Q Z l m x1)
Ass|||+ P Q Z l m (intc l .m (inj2 (inj1 x)) x1)
    = intc l m (inj1 (inj2 x)) (Ass|||+∞++ P (PI Q x) Z l m x1)
Ass|||+ P Q Z l m (intc l .m (inj2 (inj2 y)) x1)
    = intc l m (inj2 y) (Ass|||+∞++ P Q (PI Z y) l m x1)
Ass|||+ P Q Z .[] .(just ((PT P x ,, PT Q x1) ,, PT Z x2))
    (terc (x ,, (x1 ,, x2))) = terc ((x ,, x1) ,, x2)

```

```

--@END

```

```

Ass|||∞++ : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ c0)
          (Q : Process+ ∞ {lu} c1)
          (Z : Process+ ∞ {lu} c2)

```

$\rightarrow \text{Ref}\infty (((P \parallel\infty+ Q) \parallel\infty+ Z)) (\text{fmap}\infty \text{Ass}\times (P \parallel\infty+ (Q \parallel++ Z)))$
 $\text{Ass}\parallel\infty++ P Q Z l m (\text{tnode } tr) = \text{tnode } (\text{Ass}\parallel+pp (\text{forcep } P) Q Z l m tr)$

$\text{Ass}\parallel+\infty+ : \{lu : \text{LUniv}\}\{c_0 c_1 c_2 : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c_0)$
 $(Q : \text{Process}\infty \infty c_1)$
 $(Z : \text{Process}+ \infty \{lu\} c_2)$
 $\rightarrow \text{Ref}\infty (((P \parallel+\infty Q) \parallel\infty+ Z)) (\text{fmap}\infty \text{Ass}\times (P \parallel+\infty (Q \parallel\infty+ Z)))$
 $\text{Ass}\parallel+\infty+ P Q Z l m (\text{tnode } tr) = \text{tnode } (\text{Ass}\parallel+p+p P (\text{forcep } Q) Z l m tr)$

$\text{Ass}\parallel++\infty : \{lu : \text{LUniv}\}\{c_0 c_1 c_2 : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c_0)$
 $(Q : \text{Process}+ \infty \{lu\} c_1)$
 $(Z : \text{Process}\infty \infty c_2)$
 $\rightarrow \text{Ref}\infty (((P \parallel++ Q) \parallel+\infty Z)) (\text{fmap}\infty \text{Ass}\times (P \parallel+\infty (Q \parallel+\infty Z)))$
 $\text{Ass}\parallel++\infty P Q Z l m (\text{tnode } tr) = \text{tnode } (\text{Ass}\parallel+pp+ P Q (\text{forcep } Z) l m tr)$

$\text{Ass}\parallel+pp : \{lu : \text{LUniv}\}\{c_0 c_1 c_2 : \text{Choice}\}(P : \text{Process} \infty c_0)$
 $(Q : \text{Process}+ \infty \{lu\} c_1)$
 $(Z : \text{Process}+ \infty \{lu\} c_2)$
 $\rightarrow \text{Ref}+ (((P \parallel+p+ Q) \parallel++ Z)) (\text{fmap}+ \text{Ass}\times (P \parallel+p+ (Q \parallel++ Z)))$
 $\text{Ass}\parallel+pp (\text{terminate } x) Q Z l m tr = \text{Ass}\parallel-++ Q Z x l m tr$
 $\text{Ass}\parallel+pp (\text{node } x) Q Z l m tr = \text{Ass}\parallel+ x Q Z l m tr$

$\text{Ass}\parallel+p+p : \{lu : \text{LUniv}\}\{c_0 c_1 c_2 : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c_0)$
 $(Q : \text{Process} \infty c_1)$
 $(Z : \text{Process}+ \infty \{lu\} c_2)$
 $\rightarrow \text{Ref}+ (((P \parallel+p Q) \parallel++ Z)) (\text{fmap}+ \text{Ass}\times (P \parallel++ (Q \parallel+p Z)))$
 $\text{Ass}\parallel+p+p P (\text{terminate } x) Z l m tr = \text{Ass}\parallel-++ P Z x l m tr$
 $\text{Ass}\parallel+p+p P (\text{node } x) Z l m tr = \text{Ass}\parallel+ P x Z l m tr$

$\text{Ass}\parallel+pp+ : \{lu : \text{LUniv}\}\{c_0 c_1 c_2 : \text{Choice}\}(P : \text{Process}+ \infty \{lu\} c_0)$
 $(Q : \text{Process}+ \infty \{lu\} c_1)$
 $(Z : \text{Process} \infty c_2)$
 $\rightarrow \text{Ref}+ (((P \parallel++ Q) \parallel+p Z)) (\text{fmap}+ \text{Ass}\times (P \parallel++ (Q \parallel+p Z)))$
 $\text{Ass}\parallel+pp+ P Q (\text{terminate } x) l m tr = \text{Ass}\parallel+-+ P Q m l x tr$
 $\text{Ass}\parallel+pp+ P Q (\text{node } x) l m tr = \text{Ass}\parallel+ P Q x l m tr$

Ass|||++ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process+ ∞ {lu} c₁)
 (Z : Process+ ∞ {lu} c₂)

→ (x : ChoiceSet c₀)
 → (l : List (Label lu))
 → (m : Maybe maybeChoice)
 → (x₂ : Tr+ l m (fmap+ Ass× (fmap+ (λ a → a ,, x) (Q |||++ Z))))
 → Tr+ l m (fmap+ (λ a → a ,, x) Q |||++ Z)

Ass|||++ Q Z x .[] .nothing empty = empty

Ass|||++ Q Z x .(Lab Q x₁ :: l) m (extc l .m (inj₁ x₁) x₂) = extc l m (inj₁ x₁) (Ass|||∞+ (F

Ass|||++ Q Z x .(Lab Z y :: l) m (extc l .m (inj₂ y) x₂) = extc l m (inj₂ y) (Ass|||∞+

Ass|||++ Q Z x l m (intc l .m (inj₁ x₁) x₂) = intc l m (inj₁ x₁) (Ass|||∞+

Ass|||++ Q Z x l m (intc l .m (inj₂ y) x₂) = intc l m (inj₂ y) (Ass|||∞+

Ass|||++ Q Z x .[] .(just ((x ,, PT Q x₁) ,, PT Z x₂)) (terc (x₁ ,, x₂)) = terc (x₁ ,, x₂)

Ass|||+- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
 (Z : Process+ ∞ {lu} c₂)

→ (x : ChoiceSet c₁)
 → (l : List (Label lu))
 → (m : Maybe maybeChoice)
 → (tr : Tr+ l m (fmap+ Ass× (P |||++ fmap+ (λ a → a ,, x) Z)))
 → Tr+ l m (fmap+ (λ a → a ,, x) P |||++ Z)

Ass|||+- P Z x .[] .nothing empty = empty

Ass|||+- P Z x .(Lab P x₁ :: l) m (extc l .m (inj₁ x₁) x₂) = extc l m (inj₁ x₁) (Ass|||∞+ (P

Ass|||+- P Z x .(Lab Z y :: l) m (extc l .m (inj₂ y) x₂) = extc l m (inj₂ y) (Ass|||∞+

Ass|||+- P Z x l m (intc l .m (inj₁ x₁) x₂) = intc l m (inj₁ x₁) (Ass|||∞+

Ass|||+- P Z x l m (intc l .m (inj₂ y) x₂) = intc l m (inj₂ y) (Ass|||∞+

Ass|||+- P Z x .[] .(just ((PT P x₁ ,, x) ,, PT Z x₂)) (terc (x₁ ,, x₂)) = terc (x₁ ,, x₂)

Ass|||++- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
 (Q : Process+ ∞ {lu} c₁)

→ (m : Maybe maybeChoice)
 → (l : List (Label lu))
 → (x : ChoiceSet c₂)
 → (tr : Tr+ l m (fmap+ Ass× (P |||++ fmap+ (λ a → a ,, x) Q)))
 → Tr+ l m (fmap+ (λ a → a ,, x) (P |||++ Q))

Ass|||++- P Q .nothing .[] x empty = empty

Ass|||++- P Q m .(Lab P x₁ :: l) x (extc l .m (inj₁ x₁) x₂) = extc l m (inj₁ x₁) (Ass|||∞+- (P

Ass|||++- P Q m .(Lab Q y :: l) x (extc l .m (inj₂ y) x₂) = extc l m (inj₂ y) (Ass|||∞+-

```

Ass|||++- P Q m l x (intc .l .m (inj1 x1) x2) = intc l m (inj1 x1) (Ass|||∞+- (PI
Ass|||++- P Q m l x (intc .l .m (inj2 y) x2) = intc l m (inj2 y) (Ass|||+∞- P (P
Ass|||++- P Q .(just ((PT P x1 ,, PT Q x2) ,, x)) .[] x (terc (x1 ,, x2)) = terc (x1 ,, x2)

```

```

Ass|||-∞+ : {lu : LUniv}{c0 c1 c2 : Choice}{Q : Process∞ ∞ c1}
  (Z : Process+ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (x2 : Tr∞ l m (fmap∞ Ass× (fmap∞ (→,, x) (Q |||∞+ Z))))
  → Tr∞ l m (fmap∞ (→,, x) Q |||∞+ Z)
Ass|||-∞+ Q Z x l m (tnode tr) = tnode (Ass|||-p+ (forcep Q) Z x l m tr)

```

```

Ass|||∞+- : {lu : LUniv}{c0 c1 c2 : Choice}{P : Process∞ ∞ c0}
  (Z : Process+ ∞ {lu} c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (x2 : Tr∞ l m (fmap∞ Ass× ( P |||∞+ fmap+ (→,, x) Z)))
  → Tr∞ l m (fmap∞ (λ a → a ,, x) P |||∞+ Z)
Ass|||∞+- P Z x l m (tnode tr) = Ass|||p+- (forcep P) Z x l m tr

```

```

Ass|||∞+- : {lu : LUniv}{c0 c1 c2 : Choice}{P : Process∞ ∞ c0}
  (Q : Process+ ∞ {lu} c1)
  → (m : Maybe maybeChoice)
  → (l : List (Label lu))
  → (x : ChoiceSet c2)
  → (x2 : Tr∞ l m (fmap∞ Ass× ( P |||∞+ fmap+ (λ a → a ,, x) Q)))
  → Tr∞ l m (fmap∞ (λ a → a ,, x) ( P |||∞+ Q))
Ass|||∞+- P Q m l x (tnode tr) = tnode (Ass|||p+- (forcep P) Q m l x tr)

```

```

Ass|||+∞- : {lu : LUniv}{c0 c1 c2 : Choice}{P : Process+ ∞ {lu} c0}
  (Q : Process∞ ∞ c1)
  → (m : Maybe maybeChoice)
  → (l : List (Label lu))
  → (x : ChoiceSet c2)

```

```

→ (x₂ : Tr∞ l m (fmap∞ Ass× (P |||+∞ fmap∞ (λ a → a „ x) Q )))
→ Tr∞ l m (fmap∞ (λ a → a „ x) (P |||+∞ Q ))
Ass|||+∞- P Q m l x (tnode tr) = tnode (Ass|||+p- P (forcep Q) m l x tr)

```

```

Ass|||+∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
          (Z : Process∞ ∞ c₂)
→ (x : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr∞ l m (fmap∞ Ass× (P |||+∞ fmap∞ (→, - x) (Z))))
→ Tr∞ l m (fmap+ (λ a → a „ x) P |||+∞ Z)
Ass|||+∞ P Z x l m (tnode tr) = tnode (Ass|||+p P (forcep Z) x l m tr)

```

```

Ass|||-+∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process+ ∞ {lu} c₁)
          (Z : Process∞ ∞ c₂)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr∞ l m (fmap∞ Ass× (fmap∞ (→, - x) (Q |||+∞ Z))))
→ Tr∞ l m (fmap+ (→, - x) Q |||+∞ Z)
Ass|||-+∞ Q Z x l m (tnode tr) = tnode (Ass|||-+p Q (forcep Z) x l m tr)

```

```

Ass|||-p+ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process ∞ c₁)
          (Z : Process+ ∞ {lu} c₂)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr+ l m (fmap+ Ass× (fmap+ (→, - x)(Q |||p+ Z))))
→ Tr+ l m (fmap (→, - x) Q |||p+ Z)
Ass|||-p+ (terminate q) Z x l m tr = Ass|||--+ Z q x l m tr
Ass|||-p+ (node q) Z x l m tr = Ass|||-++ q Z x l m tr

```

```

Ass|||+p : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
          (Z : Process ∞ c₂)
→ (x : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)

```

```

    → (x2 : Tr+ l m (fmap+ Ass× (P |||+p fmap (→,→ x) Z)))
    → Tr+ l m (fmap+ (λ a → a „ x) P |||+p Z)
Ass|||+p P (terminate x) x1 l m x2 = Ass|||+- P m l x1 x x2
Ass|||+p P (node x) x1 l m x2 = Ass|||++ P x x1 l m x2

Ass|||+p- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
    (Q : Process ∞ c1)
    → (m : Maybe maybeChoice)
    → (l : List (Label lu))
    → (x : ChoiceSet c2)
    → (x2 : Tr+ l m (fmap+ Ass× (P |||+p fmap (λ a → a „ x) Q)))
    → Tr+ l m (fmap+ (λ a → a „ x) (P |||+p Q))
Ass|||+p- P (terminate x) m l x1 x2 = Ass|||+- P m l x x1 x2
Ass|||+p- P (node x) m l x1 x2 = Ass|||++ P x m l x1 x2

Ass|||p+- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
    (Q : Process+ ∞ {lu} c1)
    → (m : Maybe maybeChoice)
    → (l : List (Label lu))
    → (x : ChoiceSet c2)
    → (x2 : Tr+ l m (fmap+ Ass× (P |||p+ fmap+ (λ a → a „ x) Q)))
    → Tr+ l m (fmap+ (λ a → a „ x) (P |||p+ Q))
Ass|||p+- (terminate x) Q m l x1 x2 = Ass|||+- Q x x1 l m x2
Ass|||p+- (node x) Q m l x1 x2 = Ass|||++ x Q m l x1 x2

Ass|||p+ : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
    (Z : Process+ ∞ {lu} c2)
    → (x : ChoiceSet c1)
    → (l : List (Label lu))
    → (m : Maybe maybeChoice)
    → (x2 : Tr+ l m (fmap+ Ass× (P |||p+ fmap+ (→,→ x) Z)))
    → Tr l m (node (fmap (λ a → a „ x) (P) |||p+ Z))
Ass|||p+ (terminate x) Z x1 l m x2 = tnode (Ass|||++ Z x1 x l m x2)
Ass|||p+ (node x) Z x1 l m x2 = tnode (Ass|||++ x Z x1 l m x2)

Ass|||+p : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
    (Z : Process ∞ c2)
    → (x : ChoiceSet c0)

```

```

→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr+ l m (fmap+ Ass× (fmap+ (→,→ x) (Q |||+p Z))))
→ Tr+ l m (fmap+ (→,→ x) Q |||+p Z)
Ass|||+p Q (terminate x) x₁ l m tr = Ass|||+p Q x₁ x l m tr
Ass|||+p Q (node x) x₁ l m tr      = Ass|||+p Q x x₁ l m tr

```

```

Ass|||+-- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₁)
→ (x₁ : ChoiceSet c₂)
→ (x₂ : Tr+ l m (fmap+ Ass× (fmap+ (λ a → a ,, (x ,, x₁)) P)))
→ Tr+ l m (fmap+ (λ a → a ,, x₁) (fmap+ (λ a → a ,, x) P))
Ass|||+-- P .nothing .[] x x₁ empty = empty
Ass|||+-- P m .(Lab P x₂ :: l) x x₁ (extc l .m x₂ x₃) = extc l m x₂ (Ass|||∞-- (PE P x₂) m)
Ass|||+-- P m l x x₁ (intc .l .m x₂ x₃) = intc l m x₂ (Ass|||∞-- (PI P x₂) m)
Ass|||+-- P .(just ((PT P x₂ ,, x) ,, x₁)) .[] x x₁ (terc x₂) = terc x₂

```

```

Ass|||∞-- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process∞ ∞ c₀)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₁)
→ (x₁ : ChoiceSet c₂)
→ (x₃ : Tr∞ l m (fmap∞ Ass× (fmap∞ (λ a → a ,, (x ,, x₁)) P)))
→ Tr∞ l m (fmap∞ (λ a → a ,, x₁) (fmap∞ (λ a → a ,, x) P))
Ass|||∞-- P m l x x₁ tr = Ass|||p-- (forcep P) x x₁ l m tr

```

```

Ass|||--+ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Z : Process+ ∞ {lu} c₂)
→ (q : ChoiceSet c₁)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (tr : Tr+ l m (fmap+ Ass× (fmap+ (→,→ x) (fmap+ (→,→ q) Z))))
→ Tr+ l m (fmap+ (→,→ (x ,, q)) Z)
Ass|||--+ Z q x .[] .nothing empty = empty
Ass|||--+ Z q x .(Lab Z x₁ :: l) m (extc l .m x₁ x₂) = extc l m x₁ (Ass|||∞-- (P Z x₁) m)
Ass|||--+ Z q x l m (intc .l .m x₁ x₂) = intc l m x₁ (Ass|||∞-- (P Z x₁) m)
Ass|||--+ Z q x .[] .(just ((x ,, q) ,, PT Z x₁)) (terc x₁) = (terc x₁)

```

```

Ass|||+- : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
  → (x1 : ChoiceSet c0)
  → (x : ChoiceSet c2)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (tr : Tr+ l m (fmap+ Ass× (fmap+ (→,→ x1) (fmap+ (λ a → a ,, x) Q))))
  → Tr+ l m (fmap+ (λ a → a ,, x) (fmap+ (→,→ x1) Q))
Ass|||+- Q x1 x .[] .nothing empty = empty
Ass|||+- Q x1 x .(Lab Q x2 :: l) m (extc l .m x2 x3) = extc l m x2 (Ass|||∞- (PE Q x2) m l x1 x x3)
Ass|||+- Q x1 x l m (intc .l .m x2 x3) = intc l m x2 (Ass|||∞- (PI Q x2) m l x1 x x3)
Ass|||+- Q x1 x .[] .(just ((x1 ,, PT Q x2) ,, x)) (terc x2) = (terc x2)

```

```

Ass|||∞- : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ c2)
  → (q : ChoiceSet c1)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (x2 : Tr∞ l m (fmap∞ Ass× (fmap∞ (→,→ x) (fmap∞ (→,→ q) Z))))
  → Tr∞ l m (fmap∞ (→,→ (x ,, q)) Z)
Ass|||∞- Z q x l m tr = Ass|||∞-p (forcep Z) x q l m tr

```

```

Ass|||∞- : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process∞ ∞ c1)
  → (m : Maybe ((ChoiceSet c0 auxData.× ChoiceSet c1) auxData.× ChoiceSet c2))
  → (l : List (Label lu))
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (fmap∞ Ass× (fmap∞ (→,→ x) (fmap∞ (λ a → a ,, x1) Q))))
  → Tr∞ l m (fmap∞ (λ a → a ,, x1) (fmap∞ (→,→ x) Q))
Ass|||∞- Q m l x x1 tr = Ass|||∞-p (forcep Q) x x1 l m tr

```

```

Ass|||∞-p : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process ∞ c2)
  → (x : ChoiceSet c0)
  → (q : ChoiceSet c1)
  → (l : List (Label lu))

```

```

→ (m : Maybe maybeChoice)
→ (x₂ : Tr l m (fmap Ass× (fmap (→,→ x)(fmap (→,→ q) Z) )))
→ Tr l m (fmap (→,→ (x ,, q)) Z)
Ass|||--p (terminate x) x₁ q l m x₂ = lemFmap (λ x₃ → x) ((→,→ (x₁ ,, q))) (terminate (x)) l m
Ass|||--p (node x) x₁ q l m (tnode tr) = tnode (Ass|||--+ x q x₁ l m tr)

```

```

Ass|||p- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process ∞ c₁)
→ (x₁ : ChoiceSet c₀)
→ (x : ChoiceSet c₂)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₃ : Tr l m (fmap Ass× (fmap (→,→ x₁) (fmap (λ a → a ,, x) Q))))
→ Tr l m (fmap (λ a → a ,, x) (fmap (→,→ x₁) (Q)))
Ass|||p- (terminate x) x₁ x₂ l m x₃ = lemFmap ((λ x₃ → x)) (λ x₄ → (x₁ ,, x) ,, x₂) (terminate x) x₁ x₂ l m
Ass|||p- (node x) x₁ q l m (tnode tr) = tnode (Ass|||+- x x₁ q l m tr)

```

```

Ass|||p-- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process ∞ c₀)
→ (x₁ : ChoiceSet c₁)
→ (x : ChoiceSet c₂)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₃ : Tr l m (fmap Ass× (fmap (λ a → a ,, (x₁ ,, x)) P)))
→ Tr l m (fmap (λ a → a ,, x) (fmap (λ a → a ,, x₁) P))
Ass|||p-- (terminate x) x₁ x₂ l m x₃ = lemFmap (λ x₄ → x) (λ x₄ → (x ,, x₁) ,, x₂) (terminate x) x₁ x₂ l m
Ass|||p-- (node x) x₁ x₂ l m (tnode tr) = tnode (Ass|||+- x m l x₁ x₂ tr)

```

mutual

```

Ass|||+R : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process+ ∞ {lu} c₀) (Q : Process+ ∞ {lu} c₁)
→ fmap+ Ass× (P |||++ (Q |||++ Z)) ⊑+ ((P |||++ Q) |||++ Z)
Ass|||+R P Q Z .[] .nothing empty = empty
Ass|||+R P Q Z .(Lab P x :: l) m (extc l .m (inj₁ (inj₁ x)) x₁) = extc l m (inj₁ x) (Ass|||∞+ P Q Z .[] .nothing empty)
Ass|||+R P Q Z .(Lab Q y :: l) m (extc l .m (inj₁ (inj₂ y)) x₁) = extc l m (inj₂ (inj₁ y)) (Ass|||∞+ P Q Z .[] .nothing empty)

```

```

Ass||+R P Q Z .(Lab Z y :: l) m (extc l .m (inj2 y) x1)           = extc l m (inj2 (inj2 y)) (Ass||++∞R P Q Z l m (intc .l .m (inj1 (inj1 x)) x1)
Ass||+R P Q Z l m (intc .l .m (inj1 (inj1 x)) x1)                 = intc l m (inj1 x) (Ass||∞++R (PI L P Q Z l m (intc .l .m (inj1 (inj2 y)) x1)
Ass||+R P Q Z l m (intc .l .m (inj1 (inj2 y)) x1)                 = intc l m (inj2 (inj1 y)) (Ass||++∞R P Q Z l m (intc .l .m (inj2 y) x1)
Ass||+R P Q Z l m (intc .l .m (inj2 y) x1)                       = intc l m (inj2 (inj2 y)) (Ass||++∞R P Q Z l m (intc .l .m (inj2 y) x1)
Ass||+R P Q Z .[] .(just ((PT P x ,, PT Q x1) ,, PT Z x2)) (terc ((x ,, x1) ,, x2)) = terc (x ,, (x1 ,, x2))

```

```

Ass||∞++R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ c0)
  (Q : Process+ ∞ {lu} c1)
  (Z : Process+ ∞ {lu} c2)
  → Ref∞ (fmap∞ Ass× ( P |||∞+ (Q |||++ Z))) (((P |||∞+ Q) |||∞+ Z))
Ass||∞++R P Q Z l m (tnode tr) = tnode (Ass||+ppR (forcep P) Q Z l m tr)

```

```

Ass||+∞+R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  (Q : Process∞ ∞ c1)
  (Z : Process+ ∞ {lu} c2)
  → Ref∞ (fmap∞ Ass× ( P |||+∞ (Q |||∞+ Z))) (((P |||+∞ Q) |||∞+ Z))
Ass||+∞+R P Q Z l m (tnode tr) = tnode (Ass||p+pR P (forcep Q) Z l m tr)

```

```

Ass||++∞R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  (Q : Process+ ∞ {lu} c1)
  (Z : Process∞ ∞ c2)
  → Ref∞ (fmap∞ Ass× ( P |||+∞ (Q |||+∞ Z))) (((P |||++ Q) |||+∞ Z))
Ass||++∞R P Q Z l m (tnode tr) = tnode (Ass||pp+R P Q (forcep Z) l m tr)

```

```

Ass||+ppR : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ c0)
  (Q : Process+ ∞ {lu} c1)
  (Z : Process+ ∞ {lu} c2)
  → Ref+ (fmap+ Ass× ( P |||p+ (Q |||++ Z))) (((P |||p+ Q) |||++ Z))
Ass||+ppR (terminate x) Q Z l m tr = Ass||-++R Q Z x l m tr
Ass||+ppR (node x) Q Z l m tr      = Ass||+R x Q Z l m tr

```

```

Ass||p+pR : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  (Q : Process ∞ c1)
  (Z : Process+ ∞ {lu} c2)
  → Ref+ (fmap+ Ass× ( P |||++ (Q |||p+ Z))) (((P |||p+ Q) |||++ Z))
Ass||p+pR P (terminate x) Z l m tr = Ass||-++R P Z x l m tr
Ass||p+pR P (node x) Z l m tr      = Ass||+R P x Z l m tr

```

```

Ass|||pp+R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
              (Q : Process+ ∞ {lu} c1)
              (Z : Process ∞ c2)
  → Ref+ (fmap+ Ass× ( P |||++ (Q |||+p Z))) (((P |||++ Q) |||+p Z))
Ass|||pp+R P Q (terminate x) l m tr = Ass|||++-R P Q m l x tr
Ass|||pp+R P Q (node x) l m tr      = Ass|||+R P Q x l m tr

```

```

Ass|||+-+R : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
              (Z : Process+ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (x2 : Tr+ l m (fmap+ (—, —) x) Q |||++ Z))
  → Tr+ l m (fmap+ Ass× (fmap+ (—, —) x) (Q |||++ Z)))
Ass|||+-+R Q Z x .[] .nothing empty = empty
Ass|||+-+R Q Z x .(Lab Q x1 :: l) m (extc l .m (inj1 x1) x2) = extc l m (inj1 x1) (Ass|||∞-+R Q Z x2 .[] .nothing empty)
Ass|||+-+R Q Z x .(Lab Z y :: l) m (extc l .m (inj2 y) x2) = extc l m (inj2 y) (Ass|||∞-+R Q Z x2 .[] .nothing empty)
Ass|||+-+R Q Z x l m (intc .l .m (inj1 x1) x2) = intc l m (inj1 x1) (Ass|||∞-+R Q Z x2 .[] .nothing empty)
Ass|||+-+R Q Z x l m (intc .l .m (inj2 y) x2) = intc l m (inj2 y) (Ass|||∞-+R Q Z x2 .[] .nothing empty)
Ass|||+-+R Q Z x .[] .(just ((x ,, PT Q x1) ,, PT Z x2)) (terc (x1 ,, x2)) = terc (x1 ,, x2)

```

```

Ass|||+-+R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
              (Z : Process+ ∞ {lu} c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (tr : Tr+ l m (fmap+ (λ a → a ,, x) P |||++ Z))
  → Tr+ l m (fmap+ Ass× (P |||++ fmap+ (—, —) x) Z))
Ass|||+-+R P Z x .[] .nothing empty = empty
Ass|||+-+R P Z x .(Lab P x1 :: l) m (extc l .m (inj1 x1) x2) = extc l m (inj1 x1) (Ass|||∞-+R P Z x2 .[] .nothing empty)
Ass|||+-+R P Z x .(Lab Z y :: l) m (extc l .m (inj2 y) x2) = extc l m (inj2 y) (Ass|||∞-+R P Z x2 .[] .nothing empty)
Ass|||+-+R P Z x l m (intc .l .m (inj1 x1) x2) = intc l m (inj1 x1) (Ass|||∞-+R P Z x2 .[] .nothing empty)
Ass|||+-+R P Z x l m (intc .l .m (inj2 y) x2) = intc l m (inj2 y) (Ass|||∞-+R P Z x2 .[] .nothing empty)
Ass|||+-+R P Z x .[] .(just ((PT P x1 ,, x) ,, PT Z x2)) (terc (x1 ,, x2)) = terc (x1 ,, x2)

```

```

Ass|||+-+R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
              (Q : Process+ ∞ {lu} c1)

```

```

→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₂)
→ (tr : Tr+ l m (fmap+ (λ a → a „ x) (P |||++ Q)))
→ Tr+ l m (fmap+ Ass× (P |||++ fmap+ (λ a → a „ x) Q))
Ass|||++-R P Q .nothing .[] x empty = empty
Ass|||++-R P Q m .(Lab P x₁ :: l) x (extc l .m (inj₁ x₁) x₂) = extc l m (inj₁ x₁) (Ass|||∞+-R (PE P x₁) Q x₂)
Ass|||++-R P Q m .(Lab Q y :: l) x (extc l .m (inj₂ y) x₂) = extc l m (inj₂ y) (Ass|||+∞-R P (PE Q y) x₂)
Ass|||++-R P Q m l x (intc .l .m (inj₁ x₁) x₂) = intc l m (inj₁ x₁) (Ass|||∞+-R (PI P x₁) Q x₂)
Ass|||++-R P Q m l x (intc .l .m (inj₂ y) x₂) = intc l m (inj₂ y) (Ass|||+∞-R P (PI Q y) x₂)
Ass|||++-R P Q .(just ((PT P x₁ „ PT Q x₂) „ x)) .[] x (terc (x₁ „ x₂)) = terc (x₁ „ x₂)

```

```

Ass|||∞+-R : {lu : LUniv}{c₀ c₁ c₂ : Choice}{Q : Process∞ ∞ c₁}
              (Z : Process+ ∞ {lu} c₂)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr∞ l m (fmap∞ (λ a → a „ x) Q |||∞+ Z))
→ Tr∞ l m (fmap∞ Ass× (fmap∞ (λ a → a „ x) (Q |||∞+ Z)))
Ass|||∞+-R Q Z x l m (tnode tr) = tnode (Ass|||p+-R (forcep Q) Z x l m tr)

```

```

Ass|||∞+-R : {lu : LUniv}{c₀ c₁ c₂ : Choice}{P : Process∞ ∞ c₀}
              (Z : Process+ ∞ {lu} c₂)
→ (x : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr∞ l m (fmap∞ (λ a → a „ x) P |||∞+ Z))
→ Tr∞ l m (fmap∞ Ass× (P |||∞+ fmap+ (λ a → a „ x) Z))
Ass|||∞+-R P Z x l m tr = tnode (Ass|||p+-R (forcep P) Z x l m tr)

```

```

Ass|||∞+-R : {lu : LUniv}{c₀ c₁ c₂ : Choice}{P : Process∞ ∞ c₀}
              (Q : Process+ ∞ {lu} c₁)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₂)
→ (x₂ : Tr∞ l m (fmap∞ (λ a → a „ x) (P |||∞+ Q)))
→ Tr∞ l m (fmap∞ Ass× (P |||∞+ fmap+ (λ a → a „ x) Q))

```

$$\text{Ass}|||\infty+-R \ P \ Q \ m \ l \ x \ (\text{tnode } tr) = \text{tnode } (\text{Ass}|||p+-R \ (\text{forcep } P) \ Q \ m \ l \ x \ tr)$$

$$\begin{aligned} \text{Ass}|||+\infty-R & : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} \ c_0) \\ & \quad (Q : \text{Process}\infty \infty \ c_1) \\ & \rightarrow (m : \text{Maybe maybeChoice}) \\ & \rightarrow (l : \text{List } (\text{Label } lu)) \\ & \rightarrow (x : \text{ChoiceSet } c_2) \\ & \rightarrow (x_2 : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ (\lambda a \rightarrow a \text{ ,, } x) \ (P |||+\infty \ Q))) \\ & \rightarrow \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ \text{Ass}\times \ (P |||+\infty \ \text{fmap}\infty \ (\lambda a \rightarrow a \text{ ,, } x) \ Q)) \\ \text{Ass}|||+\infty-R \ P \ Q \ m \ l \ x \ (\text{tnode } tr) & = \text{tnode } (\text{Ass}|||+p-R \ P \ (\text{forcep } Q) \ m \ l \ x \ tr) \end{aligned}$$

$$\begin{aligned} \text{Ass}|||+-\infty R & : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} \ c_0) \\ & \quad (Z : \text{Process}\infty \infty \ c_2) \\ & \rightarrow (x : \text{ChoiceSet } c_1) \\ & \rightarrow (l : \text{List } (\text{Label } lu)) \\ & \rightarrow (m : \text{Maybe maybeChoice}) \\ & \rightarrow (x_2 : \text{Tr}\infty \ l \ m \ (\text{fmap}+ \ (\lambda a \rightarrow a \text{ ,, } x) \ P |||+\infty \ Z)) \\ & \rightarrow \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ \text{Ass}\times \ (P |||+\infty \ \text{fmap}\infty \ (\text{--,,-- } x) \ (Z))) \\ \text{Ass}|||+-\infty R \ P \ Z \ x \ l \ m \ (\text{tnode } tr) & = \text{tnode } (\text{Ass}|||+pR \ P \ (\text{forcep } Z) \ x \ l \ m \ tr) \end{aligned}$$

$$\begin{aligned} \text{Ass}|||-\infty R & : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (Q : \text{Process}+ \infty \{lu\} \ c_1) \\ & \quad (Z : \text{Process}\infty \infty \ c_2) \\ & \rightarrow (x : \text{ChoiceSet } c_0) \\ & \rightarrow (l : \text{List } (\text{Label } lu)) \\ & \rightarrow (m : \text{Maybe maybeChoice}) \\ & \rightarrow (x_2 : \text{Tr}\infty \ l \ m \ (\text{fmap}+ \ (\text{--,,-- } x) \ Q |||+\infty \ Z)) \\ & \rightarrow \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ \text{Ass}\times \ (\text{fmap}\infty \ (\text{--,,-- } x) \ (Q |||+\infty \ Z))) \\ \text{Ass}|||-\infty R \ Q \ Z \ x \ l \ m \ (\text{tnode } tr) & = \text{tnode } (\text{Ass}|||+pR \ Q \ (\text{forcep } Z) \ x \ l \ m \ tr) \end{aligned}$$

$$\begin{aligned} \text{Ass}|||+pR & : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (Q : \text{Process} \infty \ c_1) \\ & \quad (Z : \text{Process}+ \infty \{lu\} \ c_2) \\ & \rightarrow (x : \text{ChoiceSet } c_0) \\ & \rightarrow (l : \text{List } (\text{Label } lu)) \\ & \rightarrow (m : \text{Maybe maybeChoice}) \\ & \rightarrow (x_2 : \text{Tr}+ \ l \ m \ (\text{fmap} \ (\text{--,,-- } x) \ Q |||p+ \ Z)) \end{aligned}$$

```

→ Tr+ l m (fmap+ Ass× (fmap+ (λ a → a ,, x) (Q |||p+ Z)))
Ass|||p+R (terminate q) Z x l m tr = Ass|||p+R Z q x l m tr
Ass|||p+R (node q) Z x l m tr      = Ass|||p+R q Z x l m tr

```

```

Ass|||+-pR : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
              (Z : Process ∞ c2)
→ (x : ChoiceSet c1)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x2 : Tr+ l m (fmap+ (λ a → a ,, x) P |||+p Z))
→ Tr+ l m (fmap+ Ass× (P |||+p fmap (λ a → a ,, x) Z))
Ass|||+-pR P (terminate x) x1 l m x2 = Ass|||+-R P m l x1 x x2
Ass|||+-pR P (node x) x1 l m x2      = Ass|||+-R P x x1 l m x2

```

```

Ass|||+p-R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
              (Q : Process ∞ c1)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c2)
→ (x2 : Tr+ l m (fmap+ (λ a → a ,, x) (P |||+p Q)))
→ Tr+ l m (fmap+ Ass× (P |||+p fmap (λ a → a ,, x) Q))
Ass|||+p-R P (terminate x) m l x1 x2 = Ass|||+-R P m l x x1 x2
Ass|||+p-R P (node x) m l x1 x2      = Ass|||+-R P x m l x1 x2

```

```

Ass|||p+-R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
              (Q : Process+ ∞ {lu} c1)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c2)
→ (x2 : Tr+ l m (fmap+ (λ a → a ,, x) (P |||p+ Q)))
→ Tr+ l m (fmap+ Ass× (P |||p+ fmap+ (λ a → a ,, x) Q))
Ass|||p+-R (terminate x) Q m l x1 x2 = Ass|||+-R Q x x1 l m x2
Ass|||p+-R (node x) Q m l x1 x2      = Ass|||+-R x Q m l x1 x2

```

```

Ass|||p+R : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
              (Z : Process+ ∞ {lu} c2)
→ (x : ChoiceSet c1)
→ (l : List (Label lu))

```

```

→ (m : Maybe maybeChoice)
→ (x₂ : Tr l m (node (fmap (λ a → a ,, x) ( P |||p+ Z)))
→ Tr+ l m (fmap+ Ass× ( P |||p+ fmap+ (λ a → a ,, x) Z))
Ass|||p+R (terminate x) Z x₁ l m (tnode tr) = Ass|||--+R Z x₁ x l m tr
Ass|||p+R (node x) Z x₁ l m (tnode tr) = Ass|||+--+R x Z x₁ l m tr

```

```

Ass|||+pR : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process+ ∞ {lu} c₁)
(Z : Process ∞ c₂)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeChoice)
→ (x₂ : Tr+ l m (fmap+ (λ a → a ,, x) Q |||+p Z))
→ Tr+ l m (fmap+ Ass× (fmap+ (λ a → a ,, x) (Q |||+p Z)))
Ass|||+pR Q (terminate x) x₁ l m tr = Ass|||+--+R Q x₁ x l m tr
Ass|||+pR Q (node x) x₁ l m tr = Ass|||+--+R Q x x₁ l m tr

```

```

Ass|||+--R : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₁)
→ (x₁ : ChoiceSet c₂)
→ (x₂ : Tr+ l m (fmap+ (λ a → a ,, x₁) (fmap+ (λ a → a ,, x) P)))
→ Tr+ l m (fmap+ Ass× (fmap+ (λ a → a ,, (x ,, x₁)) P))
Ass|||+--R P .nothing .[] x x₁ empty = empty
Ass|||+--R P m .(Lab P x₂ :: l) x x₁ (extc l .m x₂ x₃) = extc l m x₂ (Ass|||∞--R (P
Ass|||+--R P m l x x₁ (intc l .m x₂ x₃) = intc l m x₂ (Ass|||∞--R (P
Ass|||+--R P .(just ((PT P x₂ ,, x) ,, x₁)) .[] x x₁ (terc x₂) = terc x₂

```

```

Ass|||∞--R : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process∞ ∞ c₀)
→ (m : Maybe maybeChoice)
→ (l : List (Label lu))
→ (x : ChoiceSet c₁)
→ (x₁ : ChoiceSet c₂)
→ (x₃ : Tr∞ l m (fmap∞ (λ a → a ,, x₁) (fmap∞ (λ a → a ,, x) P)))
→ Tr∞ l m (fmap∞ Ass× (fmap∞ (λ a → a ,, (x ,, x₁)) P))
Ass|||∞--R P m l x x₁ tr = Ass|||p--R (forcep P) x x₁ l m tr

```

```

Ass||--+R : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process+ ∞ {lu} c2)
  → (q : ChoiceSet c1)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (tr : Tr+ l m (fmap+ (→,→ (x ,, q)) Z))
  → Tr+ l m (fmap+ Ass× (fmap+ (→,→ x) (fmap+ (→,→ q) Z)))

```

```

Ass||--+R Z q x .[] .nothing empty = empty
Ass||--+R Z q x .(Lab Z x1 :: l) m (extc l .m x1 x2) = extc l m x1 (Ass||--∞R (PE Z x1) q x l m x2)
Ass||--+R Z q x l m (intc .l .m x1 x2) = intc l m x1 (Ass||--∞R (PI Z x1) q x l m x2)
Ass||--+R Z q x .[] .(just ((x ,, q) ,, PT Z x1)) (terc x1) = (terc x1)

```

```

Ass||-+-R : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
  → (x1 : ChoiceSet c0)
  → (x : ChoiceSet c2)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (tr : Tr+ l m (fmap+ (λ a → a ,, x) (fmap+ (→,→ x1) Q)))
  → Tr+ l m (fmap+ Ass× (fmap+ (→,→ x1) (fmap+ (λ a → a ,, x) Q)))
Ass||-+-R Q x1 x .[] .nothing empty = empty
Ass||-+-R Q x1 x .(Lab Q x2 :: l) m (extc l .m x2 x3) = extc l m x2 (Ass||--∞R (PE Q x2) m x3)
Ass||-+-R Q x1 x l m (intc .l .m x2 x3) = intc l m x2 (Ass||--∞R (PI Q x2) m x3)
Ass||-+-R Q x1 x .[] .(just ((x1 ,, PT Q x2) ,, x)) (terc x2) = (terc x2)

```

```

Ass||--∞R : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ c2)
  → (q : ChoiceSet c1)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeChoice)
  → (x2 : Tr∞ l m (fmap∞ (→,→ (x ,, q)) Z))
  → Tr∞ l m (fmap∞ Ass× (fmap∞ (→,→ x) (fmap∞ (→,→ q) Z)))
Ass||--∞R Z q x l m tr = Ass||--pR (forcep Z) x q l m tr

```

```

Ass||-∞-R : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process∞ ∞ c1)
  → (m : Maybe maybeChoice)
  → (l : List (Label lu))
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (fmap∞ (λ a → a ,, x1) (fmap∞ (→,→ x) Q)))
  → Tr∞ l m (fmap∞ Ass× (fmap∞ (→,→ x) (fmap∞ (λ a → a ,, x1) Q)))

```

$\text{Ass}|||-\infty\text{-R } Q \ m \ l \ x \ x_1 \ tr = \text{Ass}|||-\text{p-R } (\text{forcep } Q) \ x \ x_1 \ l \ m \ tr$

$\text{Ass}|||-\text{p-R} : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (Z : \text{Process } \infty \ c_2)$

$\rightarrow (x : \text{ChoiceSet } c_0)$

$\rightarrow (q : \text{ChoiceSet } c_1)$

$\rightarrow (l : \text{List } (\text{Label } lu))$

$\rightarrow (m : \text{Maybe maybeChoice})$

$\rightarrow (x_2 : \text{Tr } l \ m \ (\text{fmap } (_,_ -) (x _,_ q)) \ Z))$

$\rightarrow \text{Tr } l \ m \ (\text{fmap } \text{Ass} \times (\text{fmap } (_,_ -) x) (\text{fmap } (_,_ -) q) \ Z))$

$\text{Ass}|||-\text{p-R } (\text{terminate } x) \ x_1 \ q \ l \ m \ x_2 = \text{lemFmapR } (\lambda x_3 \rightarrow x) ((_,_ -) (x_1 _,_ q)) (\text{terminate } (x))$

$\text{Ass}|||-\text{p-R } (\text{node } x) \ x_1 \ q \ l \ m \ (\text{tnode } tr) = \text{tnode } (\text{Ass}|||-\text{+R } x \ q \ x_1 \ l \ m \ tr)$

$\text{Ass}|||-\text{p-R} : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (Q : \text{Process } \infty \ c_1)$

$\rightarrow (x_1 : \text{ChoiceSet } c_0)$

$\rightarrow (x _ : \text{ChoiceSet } c_2)$

$\rightarrow (l _ : \text{List } (\text{Label } lu))$

$\rightarrow (m _ : \text{Maybe maybeChoice})$

$\rightarrow (x_3 : \text{Tr } l \ m \ (\text{fmap } (\lambda a \rightarrow a _,_ x) (\text{fmap } (_,_ -) x_1) (Q))))$

$\rightarrow \text{Tr } l \ m \ (\text{fmap } \text{Ass} \times (\text{fmap } (_,_ -) x_1) (\text{fmap } (\lambda a \rightarrow a _,_ x) Q)))$

$\text{Ass}|||-\text{p-R } (\text{terminate } x) \ x_1 \ x_2 \ l \ m \ x_3 = \text{lemFmapR } ((\lambda x_3 \rightarrow x)) (\lambda x_4 \rightarrow (x_1 _,_ x) _,_ x_2) (\text{termina$

$\text{Ass}|||-\text{p-R } (\text{node } x) \ x_1 \ x_2 \ l \ m \ (\text{tnode } tr) = \text{tnode } (\text{Ass}|||-\text{+R } x \ x_1 \ x_2 \ l \ m \ tr)$

$\text{Ass}|||-\text{p-R} : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process } \infty \ c_0)$

$\rightarrow (x_1 : \text{ChoiceSet } c_1)$

$\rightarrow (x _ : \text{ChoiceSet } c_2)$

$\rightarrow (l _ : \text{List } (\text{Label } lu))$

$\rightarrow (m _ : \text{Maybe maybeChoice})$

$\rightarrow (x_3 : \text{Tr } l \ m \ (\text{fmap } (\lambda a \rightarrow a _,_ x) (\text{fmap } (\lambda a \rightarrow a _,_ x_1) P))))$

$\rightarrow \text{Tr } l \ m \ (\text{fmap } \text{Ass} \times (\text{fmap } (\lambda a \rightarrow a _,_ (x_1 _,_ x)) P))$

$\text{Ass}|||-\text{p-R } (\text{terminate } x) \ x_1 \ x_2 \ l \ m \ x_3 = \text{lemFmapR } (\lambda x_4 \rightarrow x) (\lambda x_4 \rightarrow (x _,_ x_1) _,_ x_2) (\text{termina$

$\text{Ass}|||-\text{p-R } (\text{node } x) \ x_1 \ x_2 \ l \ m \ (\text{tnode } tr) = \text{tnode } (\text{Ass}|||-\text{+R } x \ m \ l \ x_1 \ x_2 \ tr)$

$--@BEGIN@EqAssIntDef$

$\equiv|||+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\}$

$(P : \text{Process}+ \infty \ \{lu\} \ c_0)$

$(Q : \text{Process}+ \infty \ \{lu\} \ c_1)$

$(Z : \text{Process}+ \infty \ \{lu\} \ c_2)$

```

      → ((P |||++ Q) |||++ Z) ≡+ (fmap+ Ass× (P |||++ (Q |||++ Z)))

--@END

--@BEGIN@EqAssIntDefProof

≡|||+ P Q Z = Ass|||+ P Q Z , Ass|||+R P Q Z

--@END

```

A.71 proofAssExt.agda

```

--@PREFIX@mainproofAssExt

module proofAssExt where

open import process
open import TraceWithoutSize
open import Size
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.Product
open import Data.List
open import Data.Sum
open import Data.Fin
open import renamingResult
open import RefWithoutSize
open import dataAuxFunction
open import lemFmap
open import externalChoice
open import addTick
open import Data.Nat
open import internalChoice
open import Data.String
open import traceEquivalence
open import Data.Product

```

open import labelUniv

maybeExtC : {c₀ c₁ c₂ : Choice} → Set
 maybeExtC {c₀} {c₁} {c₂} = ((ChoiceSet c₀ ⊔ ChoiceSet c₁) ⊔ ChoiceSet c₂)

mutual

--@BEGIN@AssExDef

A□+ : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process+ ∞ {lu} c₀)
 (Q : Process+ ∞ {lu} c₁)
 (Z : Process+ ∞ {lu} c₂)
 → ((P □++ Q) □++ Z) ⊑+ fmap+ Ass⊔r (P □++ (Q □++ Z))

A□+ P Q Z .[] .nothing empty = empty

A□+ P Q Z .(Lab P x :: l) m (extc l .m (inj₁ x) x₁) =
 let

x' : Tr∞ l m (fmap∞ Ass⊔r (fmap∞ inj₁ (PE P x)))
 x' = x₁

x₁' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj₁) ((PE P x)))
 x₁' = lemFmap∞ inj₁ Ass⊔r (PE P x) l m x'

x₂' : Tr∞ l m (fmap∞ inj₁ (fmap∞ inj₁ (PE P x)))
 x₂' = lemFmap∞R inj₁ inj₁ (PE P x) l m x₁'

in extc l m (inj₁ (inj₁ x)) x₂'

A□+ P Q Z .(Lab Q x :: l) m (extc l .m (inj₂ (inj₁ x)) x₁) =
 let

x₁'' : Tr∞ l m (fmap∞ Ass⊔r
 (fmap∞ inj₂ (fmap∞ inj₁ (PE Q x))))
 x₁'' = x₁

x₁' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj₂) (fmap∞ inj₁ (PE Q x)))
 x₁' = lemFmap∞ inj₂ Ass⊔r (fmap∞ inj₁ (PE Q x)) l m x₁''

x₂' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj₂ ∘ inj₁) (PE Q x))
 x₂' = lemFmap∞ inj₁ (Ass⊔r ∘ inj₂) (PE Q x) l m x₁'

x₃' : Tr∞ l m (fmap∞ inj₁ (fmap∞ inj₂ (PE Q x)))
 x₃' = lemFmap∞R inj₂ inj₁ (PE Q x) l m x₂'

$\text{in extc } l \ m \ (\text{inj}_1 \ (\text{inj}_2 \ x)) \ x_3'$

$\text{A}\square+ \ P \ Q \ Z \ .(\text{Lab } Z \ y :: l) \ m \ (\text{extc } l \ .m \ (\text{inj}_2 \ (\text{inj}_2 \ y)) \ x_1) =$
 $\text{extc } l \ m \ (\text{inj}_2 \ y) \ (\text{lemFmap}\infty \ \text{inj}_2 \ (\text{Ass}\uplus r \circ \text{inj}_2) \ (\text{PE } Z \ y))$
 $l \ m \ (\text{lemFmap}\infty \ \text{inj}_2 \ \text{Ass}\uplus r \ (\text{fmap}\infty \ \text{inj}_2 \ (\text{PE } Z \ y)) \ l \ m \ x_1))$
 $\text{A}\square+ \ P \ Q \ Z \ l \ m \ (\text{intc } .l \ .m \ (\text{inj}_1 \ x) \ x_1) =$
 $\text{intc } l \ m \ (\text{inj}_1 \ (\text{inj}_1 \ x)) \ (\text{A}\square\infty++ \ (\text{PI } P \ x) \ Q \ Z \ l \ m \ x_1)$
 $\text{A}\square+ \ P \ Q \ Z \ l \ m \ (\text{intc } .l \ .m \ (\text{inj}_2 \ (\text{inj}_1 \ x)) \ x_1) =$
 $\text{intc } l \ m \ (\text{inj}_1 \ (\text{inj}_2 \ x)) \ (\text{A}\square+\infty+ \ P \ (\text{PI } Q \ x) \ Z \ l \ m \ x_1)$
 $\text{A}\square+ \ P \ Q \ Z \ l \ m \ (\text{intc } .l \ .m \ (\text{inj}_2 \ (\text{inj}_2 \ y)) \ x_1) =$
 $\text{intc } l \ m \ (\text{inj}_2 \ y) \ (\text{A}\square++\infty \ P \ Q \ (\text{PI } Z \ y) \ l \ m \ x_1)$
 $\text{A}\square+ \ P \ Q \ Z \ .[] \ .(\text{just } (\text{inj}_1 \ (\text{inj}_1 \ (\text{PT } P \ x))))(\text{terc } (\text{inj}_1 \ x)) =$
 $\text{terc } (\text{inj}_1 \ (\text{inj}_1 \ x))$
 $\text{A}\square+ \ P \ Q \ Z \ .[] \ .(\text{just } (\text{inj}_1 \ (\text{inj}_2 \ (\text{PT } Q \ x))))$
 $(\text{terc } (\text{inj}_2 \ (\text{inj}_1 \ x))) = \text{terc } (\text{inj}_1 \ (\text{inj}_2 \ x))$
 $\text{A}\square+ \ P \ Q \ Z \ .[] \ .(\text{just } (\text{inj}_2 \ (\text{PT } Z \ y))) \ (\text{terc } (\text{inj}_2 \ (\text{inj}_2 \ y))) =$
 $\text{terc } (\text{inj}_2 \ y)$

--@END

$\text{A}\square\infty++ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} \ c_0)$
 $(Q : \text{Process}+ \infty \{lu\} \ c_1)$
 $(Z : \text{Process}+ \infty \{lu\} \ c_2)$
 $\rightarrow \text{Ref}\infty \ (((P \ \square\infty++ \ Q) \ \square\infty++ \ Z))$
 $(\text{fmap}\infty \ \text{Ass}\uplus r \ (P \ \square\infty++ \ (Q \ \square++ \ Z)))$
 $\text{A}\square\infty++ \ P \ Q \ Z \ l \ m \ (\text{tnode } tr) = \text{tnode} \ (\text{A}\square\text{pp} \ (\text{forcep } P) \ Q \ Z \ l \ m \ tr)$

$\text{A}\square+\infty+ : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} \ c_0)$
 $(Q : \text{Process}\infty \infty \{lu\} \ c_1)$
 $(Z : \text{Process}+ \infty \{lu\} \ c_2)$
 $\rightarrow \text{Ref}\infty \ (((P \ \square+\infty+ \ Q) \ \square\infty++ \ Z)) \ (\text{fmap}\infty \ \text{Ass}\uplus r \ (P \ \square+\infty+ \ (Q \ \square\infty++ \ Z)))$
 $\text{A}\square+\infty+ \ P \ Q \ Z \ l \ m \ (\text{tnode } tr) = \text{tnode} \ (\text{A}\square\text{p}+\text{p} \ P \ (\text{forcep } Q) \ Z \ l \ m \ tr)$

$\text{A}\square++\infty : \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} \ c_0)$
 $(Q : \text{Process}+ \infty \{lu\} \ c_1)$
 $(Z : \text{Process}\infty \infty \{lu\} \ c_2)$
 $\rightarrow \text{Ref}\infty \ (((P \ \square++ \ Q) \ \square+\infty+ \ Z)) \ (\text{fmap}\infty \ \text{Ass}\uplus r \ (P \ \square++\infty \ (Q \ \square+\infty+ \ Z)))$
 $\text{A}\square++\infty \ P \ Q \ Z \ l \ m \ (\text{tnode } tr) = \text{tnode} \ (\text{A}\square\text{pp}+ \ P \ Q \ (\text{forcep } Z) \ l \ m \ tr)$

```

A□+pp : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
      (Q : Process+ ∞ {lu} c1)
      (Z : Process+ ∞ {lu} c2)
      → Ref+ (((P □p++ Q) □++ Z)) (fmap+ Ass⊔r ( P □p++ (Q □++ Z)))
A□+pp (terminate x) Q Z l m tr = A□-++ Q Z x l m tr
A□+pp (node x) Q Z l m tr      = A□+ x Q Z l m tr

```

```

A□p+p : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Q : Process ∞ c1)
      (Z : Process+ ∞ {lu} c2)
      → Ref+ (((P □+p+ Q) □++ Z)) (fmap+ Ass⊔r ( P □++ (Q □p++ Z)))
A□p+p P (terminate x) Z l m tr = A□+-+ P Z x l m tr
A□p+p P (node x) Z l m tr      = A□+ P x Z l m tr

```

```

A□pp+ : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
      (Z : Process ∞ c2)
      → Ref+ (((P □++ Q) □+p+ Z)) (fmap+ Ass⊔r ( P □++ (Q □+p+ Z)))
A□pp+ P Q (terminate x) l m tr = A□+-+ P Q m l x tr
A□pp+ P Q (node x) l m tr      = A□+ P Q x l m tr

```

```

A□++- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
      → (m : Maybe maybeExtC)
      → (l : List (Label lu))
      → (x : ChoiceSet c2)
      → (tr : Tr+ l m (fmap+ Ass⊔r (P □++ addTimed✓+ (inj2 x) (fmap+ inj1 Q))
      → Tr+ l m (addTimed✓+ (inj2 x) (fmap+ inj1 (P □++ Q))))
A□++- P Q .nothing .[] x empty = empty
A□++- P Q m .(Lab P x1 :: l) x (extc l .m (inj1 x1) x2) = let

```

```

x' :      Tr∞ l m (fmap∞ As
x' = x2

```

```

x1' :      Tr∞ l m (fmap∞

```



$$x_1' = \text{lemFmap}\infty \text{ inj}_1 \text{ Ass}\oplus\text{r} (\text{PE } P \ x_1)$$

$$x_2' : \quad \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \text{ inj}_1 \ (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } P \ x_1) \ l \ m))$$

$$x_2' = \text{lemFmap}\infty\text{R} \text{ inj}_1 \text{ inj}_1 (\text{PE } P \ x_1) \ l \ m$$

$$\text{in extc } l \ m \ (\text{inj}_1 \ x_1) \ x_2'$$

$$\text{A}\square++- \ P \ Q \ m \ .(\text{Lab } Q \ y :: l) \ x \ (\text{extc } l \ .m \ (\text{inj}_2 \ y) \ x_2) = \text{let}$$

$$x' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{fmap}\infty \text{ inj}_2 \ (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_2))$$

$$x' = x_2$$

$$x_1' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty (\text{Ass}\oplus\text{r} \circ \text{inj}_2) (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_1)$$

$$x_1' = \text{lemFmap}\infty \text{ inj}_2 \text{ Ass}\oplus\text{r} (\text{fmap}\infty \text{ inj}_1 \ (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_1)$$

$$x_2' : \quad \text{Tr}\infty \ l \ m \ (\text{fmap}\infty (\text{Ass}\oplus\text{r} \circ \text{inj}_2) (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_2)$$

$$x_2' = \text{lemFmap}\infty \text{ inj}_1 (\text{Ass}\oplus\text{r} \circ \text{inj}_2) (\text{PE } Q \ y) \ l \ m$$

$$x_3' : \quad \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \text{ inj}_1 (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_3)$$

$$x_3' = \text{lemFmap}\infty\text{R} \text{ inj}_2 \text{ inj}_1 (\text{PE } Q \ y) \ l \ m$$

$$\text{in extc } l \ m \ (\text{inj}_2 \ y) \ x_3'$$

$$\text{A}\square++- \ P \ Q \ m \ l \ x \ (\text{intc } .l \ .m \ (\text{inj}_1 \ x_1) \ x_2) = \text{intc } l \ m \ (\text{inj}_1 \ x_1) \ (\text{A}\square\infty+- \ (\text{PI } P \ x_1) \ Q \ m \ l \ x \ x_2)$$

$$\text{A}\square++- \ P \ Q \ m \ l \ x \ (\text{intc } .l \ .m \ (\text{inj}_2 \ y) \ x_2) = \text{intc } l \ m \ (\text{inj}_2 \ y) \ (\text{A}\square+\infty- \ P \ (\text{PI } Q \ y) \ m \ l \ x \ x_2)$$

$$\text{A}\square++- \ P \ Q \ .(\text{just } (\text{inj}_1 \ (\text{inj}_1 \ (\text{PT } P \ x_1)))) \ .[] \ x \ (\text{terc } (\text{inj}_1 \ x_1)) = \text{terc } (\text{inj}_2 \ (\text{inj}_1 \ x_1))$$

$$\text{A}\square++- \ P \ Q \ .(\text{just } (\text{inj}_2 \ x)) \ .[] \ x \ (\text{terc } (\text{inj}_2 \ (\text{inj}_1 \ x_1))) = \text{terc } (\text{inj}_1 \ x_1)$$

$$\text{A}\square++- \ P \ Q \ .(\text{just } (\text{inj}_1 \ (\text{inj}_2 \ (\text{PT } Q \ y)))) \ .[] \ x \ (\text{terc } (\text{inj}_2 \ (\text{inj}_2 \ y))) = \text{terc } (\text{inj}_2 \ (\text{inj}_2 \ y))$$

$$\text{A}\square-++ : \{lu : \text{LUniv}\}\{c_0 \ c_1 \ c_2 : \text{Choice}\}(Q : \text{Process+} \infty \{lu\} \ c_1)$$

$$(Z : \text{Process+} \infty \{lu\} \ c_2)$$

$$\rightarrow (x : \text{ChoiceSet } c_0)$$

$$\rightarrow (l : \text{List } (\text{Label } lu))$$

$$\rightarrow (m : \text{Maybe } ((\text{ChoiceSet } c_0 \uplus \text{ChoiceSet } c_1) \uplus \text{ChoiceSet } c_2))$$

$$\rightarrow (tr : \text{Tr+ } l \ m \ (\text{fmap+ } \text{Ass}\oplus\text{r} \ (\text{addTimed}\check{+} \ (\text{inj}_1 \ x) \ (\text{fmap+ } \text{inj}_2 \ (Q \ \square++ \ Z))))$$

$$\rightarrow \text{Tr+ } l \ m \ (\text{addTimed}\check{+} \ (\text{inj}_1 \ x) \ (\text{fmap+ } \text{inj}_2 \ Q) \ \square++ \ Z)$$

$$\text{A}\square-++ \ Q \ Z \ x \ .[] \ .\text{nothing empty} = \text{empty}$$

$$\text{A}\square-++ \ Q \ Z \ x \ .(\text{Lab } Q \ x_1 :: l) \ m \ (\text{extc } l \ .m \ (\text{inj}_1 \ x_1) \ x_2) = \text{let}$$

$$x' : \quad \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{fmap}\infty \text{ Ass}\oplus\text{r} (\text{PE } Q \ y) \ l \ m) \ x_2))$$


$$x' = x_2$$

$$\begin{aligned} x_1' &: \text{Tr} \infty l m (\text{fmap} \infty \text{ Ass} \oplus r) \\ x_1' &= \text{lemFmap} \infty \text{ inj}_2 \text{ Ass} \oplus r (f) \end{aligned}$$

$$\begin{aligned} x_2' &: \text{Tr} \infty l m (\text{fmap} \infty (\text{Ass} \oplus r)) \\ x_2' &= \text{lemFmap} \infty \text{ inj}_1 (\text{Ass} \oplus r \circ f) \end{aligned}$$

$$\begin{aligned} x_3' &: \text{Tr} \infty l m (\text{fmap} \infty \text{ Ass} \oplus r) \\ x_3' &= \text{lemFmap} \infty \text{R inj}_2 \text{ inj}_1 (\text{PT} Q x_1) \end{aligned}$$

$$\text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } . (\text{Lab } Z \text{ } y \text{ } :: l) \text{ } m \text{ } (\text{extc } l \text{ } . m \text{ } (\text{inj}_2 \text{ } y) \text{ } x_2) = \text{let } \text{in extc } l \text{ } m \text{ } (\text{inj}_1 \text{ } x_1) \text{ } x_3'$$

$$\begin{aligned} x' &: \text{Tr} \infty l m (\text{fmap} \infty \text{ Ass} \oplus r) \\ x' &= x_2 \end{aligned}$$

$$\begin{aligned} x_1' &: \text{Tr} \infty l m (\text{fmap} \infty \text{ Ass} \oplus r) \\ x_1' &= \text{lemFmap} \infty \text{ inj}_2 \text{ Ass} \oplus r (f) \end{aligned}$$

$$\begin{aligned} x_2' &: \text{Tr} \infty l m (\text{fmap} \infty \text{ Ass} \oplus r) \\ x_2' &= \text{lemFmap} \infty \text{ inj}_2 (\text{Ass} \oplus r \circ f) \end{aligned}$$

$$\begin{aligned} & \text{in extc } l \text{ } m \text{ } (\text{inj}_2 \text{ } y) \text{ } x_2' \\ \text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } l \text{ } m \text{ } (\text{intc } . l \text{ } . m \text{ } (\text{inj}_1 \text{ } x_1) \text{ } x_2) &= \text{intc } l \text{ } m \text{ } (\text{inj}_1 \text{ } x_1) (\text{A}\square\text{--}\infty+ (\text{PI } Q \text{ } x_1) \text{ } Z \text{ } x \text{ } l \text{ } m \text{ } x_2) \\ \text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } l \text{ } m \text{ } (\text{intc } . l \text{ } . m \text{ } (\text{inj}_2 \text{ } y) \text{ } x_2) &= \text{intc } l \text{ } m \text{ } (\text{inj}_2 \text{ } y) (\text{A}\square\text{--}\infty+ Q (\text{PI } Z \text{ } x_1)) \\ \text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } . [] \text{ } . (\text{just } (\text{inj}_1 \text{ } (\text{inj}_1 \text{ } x))) \text{ } (\text{terc } (\text{inj}_1 \text{ } x_1)) &= \text{terc } (\text{inj}_1 \text{ } (\text{inj}_1 \text{ } x_1)) \\ \text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } . [] \text{ } . (\text{just } (\text{inj}_1 \text{ } (\text{inj}_2 \text{ } (\text{PT } Q \text{ } x_1)))) \text{ } (\text{terc } (\text{inj}_2 \text{ } (\text{inj}_1 \text{ } x_1))) &= \text{terc } (\text{inj}_1 \text{ } (\text{inj}_2 \text{ } x_1)) \\ \text{A}\square\text{--}++ \text{ } Q \text{ } Z \text{ } x \text{ } . [] \text{ } . (\text{just } (\text{inj}_2 \text{ } (\text{PT } Z \text{ } y))) \text{ } (\text{terc } (\text{inj}_2 \text{ } (\text{inj}_2 \text{ } y))) &= \text{terc } (\text{inj}_2 \text{ } y) \end{aligned}$$

$$\text{A}\square\text{--}++ : \{lu : \text{LUniv}\} \{c_0 \text{ } c_1 \text{ } c_2 : \text{Choice}\} (P : \text{Process} + \infty \{lu\} c_0) \\ (Z : \text{Process} + \infty \{lu\} c_2)$$

$$\rightarrow (x : \text{ChoiceSet } c_1)$$

$$\rightarrow (l : \text{List } (\text{Label } lu))$$

$$\rightarrow (m : \text{Maybe maybeExtC})$$

$$\rightarrow (tr : \text{Tr} + l m (\text{fmap} + \text{Ass} \oplus r) (P \square++ \text{ addTimed} \checkmark + (\text{inj}_1 \text{ } x) (\text{fmap} + \text{inj}_2 \text{ } Z))$$

$$\rightarrow \text{Tr} + l m (\text{addTimed} \checkmark + (\text{inj}_2 \text{ } x) (\text{fmap} + \text{inj}_1 \text{ } P) \square++ Z)$$

$$\text{A}\square\text{--}++ \text{ } P \text{ } Z \text{ } x \text{ } . [] \text{ } . \text{nothing empty} = \text{empty}$$

$$\text{A}\square\text{--}++ \text{ } P \text{ } Z \text{ } x \text{ } . (\text{Lab } P \text{ } x_1 \text{ } :: l) \text{ } m \text{ } (\text{extc } l \text{ } . m \text{ } (\text{inj}_1 \text{ } x_1) \text{ } x_2) = \text{let}$$

```

A□∞+- : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process∞ ∞ {lu} c₀)
      (Q : Process+ ∞ {lu} c₁)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c₂)
  → (x₂ : Tr∞ l m (fmap∞ Ass⊔r (P □∞++ addTimed✓+ (inj₂ x) (fmap+ inj₁
  → Tr∞ l m (addTimed✓∞ (inj₂ x) (fmap∞ inj₁ (P □∞++ Q))))
A□∞+- P Q m l x (tnode tr) = tnode (A□p+- (forcep P) Q m l x tr)

```

```

A□-∞+ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Q : Process∞ ∞ {lu} c₁)
      (Z : Process+ ∞ {lu} c₂)
  → (x : ChoiceSet c₀)
  → (l : List (Label lu))
  → (m : Maybe ((ChoiceSet c₀ ⊔ ChoiceSet c₁) ⊔ ChoiceSet c₂))
  → (x₂ : Tr∞ l m (fmap∞ Ass⊔r (addTimed✓∞ (inj₁ x) (fmap∞ inj₂ (Q □∞+
  → Tr∞ l m (addTimed✓∞ (inj₁ x) (fmap∞ inj₂ Q) □∞++ Z)
A□-∞+ Q Z x l m (tnode tr) = A□p+- (forcep Q) Z x l m tr

```

```

A□∞-+ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process∞ ∞ {lu} c₀)
      (Z : Process+ ∞ {lu} c₂)
  → (x : ChoiceSet c₁)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x₂ : Tr∞ l m (fmap∞ Ass⊔r (P □∞++ addTimed✓+ (inj₁ x) (fmap+ i
  → Tr∞ l m (addTimed✓∞ (inj₂ x) (fmap∞ inj₁ (P)) □∞++ Z)
A□∞-+ P Z x l m (tnode tr) = A□p+- (forcep P) Z x l m tr

```

```

A□+-∞ : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)
      (Z : Process∞ ∞ {lu} c₂)
  → (x : ChoiceSet c₁)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x₂ : Tr∞ l m (fmap∞ Ass⊔r (P □+-∞+ addTimed✓∞ (inj₁ x) (fmap∞ inj
  → Tr∞ l m (addTimed✓+ (inj₂ x) (fmap+ inj₁ P) □+-∞+ Z)
A□+-∞ P Z x l m (tnode tr) = A□p+- P (forcep Z) x l m tr

```

```

A□-+∞ : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
      (Z : Process∞ ∞ {lu} c2)
      → (x : ChoiceSet c0)
      → (l : List (Label lu))
      → (m : Maybe maybeExtC)
      → (x2 : Tr∞ l m (fmap∞ Ass⊔r (addTimed✓∞ (inj1 x) (fmap∞ inj2 (Q □+∞+ Z)))))
      → Tr∞ l m (addTimed✓+ (inj1 x) (fmap+ inj2 Q) □+∞+ Z)
A□-+∞ Q Z x l m (tnode tr) = tnode ( A□-+p Q (forcep Z) x l m tr)

```

```

A□p+- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
      (Q : Process+ ∞ {lu} c1)
      → (m : Maybe maybeExtC)
      → (l : List (Label lu))
      → (x : ChoiceSet c2)
      → (x2 : Tr+ l m (fmap+ Ass⊔r (P □p++ addTimed✓+ (inj2 x) (fmap+ inj1 Q)))))
      → Tr+ l m ((addTimed✓+ (inj2 x) (fmap+ inj1 (P □p++ Q)))))
A□p+- (terminate x) Q m l x1 x2 = A□-+- Q x1 x l m x2
A□p+- (node x) Q m l x1 x2 = A□++- x Q m l x1 x2

```

```

A□+p- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Q : Process ∞ c1)
      → (m : Maybe maybeExtC)
      → (l : List (Label lu))
      → (x : ChoiceSet c2)
      → (x2 : Tr+ l m (fmap+ Ass⊔r (P □+p+ addTimed✓ (inj2 x) (fmap inj1 Q)))))
      → Tr+ l m ((addTimed✓+ (inj2 x) (fmap+ inj1 (P □+p+ Q)))))
A□+p- P (terminate x) m l x1 x2 = A□+-- P m l x1 x (A□+--s P m l x1 x2)
A□+p- P (node x) m l x1 x2 = A□++- P x m l x1 x2

```

```

A□+-p : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Z : Process ∞ c2)
      → (x : ChoiceSet c1)
      → (l : List (Label lu))
      → (m : Maybe maybeExtC)
      → (x2 : Tr+ l m (fmap+ Ass⊔r (P □+p+ addTimed✓ (inj1 x) (fmap inj2 Z)))))
      → Tr l m (node (addTimed✓+ (inj2 x) (fmap+ inj1 P) □+p+ Z))
A□+-p P (terminate x) x1 l m x2 = tnode (A□+-- P m l x1 x2) --

```

$$A\Box+-p \quad P \text{ (node } x) \ x_1 \ l \ m \ x_2 \quad = \text{tnode } (A\Box++ P \ x \ x_1 \ l \ m \ x_2)$$

$$\begin{aligned} A\Box+p &: \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} (Q : Process+ \infty \{lu\} \ c_1) \\ &\quad (Z : Process+ \infty \ c_2) \\ &\rightarrow (x : ChoiceSet \ c_0) \\ &\rightarrow (l : List (Label \ lu)) \\ &\rightarrow (m : Maybe maybeExtC) \\ &\rightarrow (tr : Tr+ \ l \ m \ ((fmap+ Ass\uplus r \ (addTimed\checkmark + (inj_1 \ x) \ (fmap+ inj_2 \ (Q \ \Box+p \ Z))) \\ &\rightarrow Tr+ \ l \ m \ ((addTimed\checkmark + (inj_1 \ x) \ (fmap+ inj_2 \ Q) \ \Box+p \ Z))) \end{aligned}$$

$$A\Box+p \ Q \ (\text{terminate } x) \ x_1 \ l \ m \ tr = A\Box+- Q \ x \ x_1 \ l \ m \ tr \ \text{--}$$

$$A\Box+p \ Q \ (\text{node } x) \ x_1 \ l \ m \ tr = A\Box++ Q \ x \ x_1 \ l \ m \ tr$$

$$\begin{aligned} A\Box p+ &: \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} (P : Process+ \infty \ c_0) \\ &\quad (Z : Process+ \infty \{lu\} \ c_2) \\ &\rightarrow (x : ChoiceSet \ c_1) \\ &\rightarrow (l : List (Label \ lu)) \\ &\rightarrow (m : Maybe maybeExtC) \\ &\rightarrow (x_2 : Tr+ \ l \ m \ (fmap+ Ass\uplus r \ (P \ \Box p++ addTimed\checkmark + (inj_1 \ x) \ (fmap+ inj_2 \ Z))) \\ &\rightarrow Tr \ l \ m \ (\text{node } (addTimed\checkmark \ (inj_2 \ x) \ (fmap \ inj_1 \ P) \ \Box p++ Z))) \end{aligned}$$

$$A\Box p+ \ (\text{terminate } x) \ Z \ x_1 \ l \ m \ x_2 = A\Box--+_s \ Z \ x \ x_1 \ l \ m \ x_2 \ \text{--}$$

$$A\Box p+ \ (\text{node } x) \ Z \ x_1 \ l \ m \ x_2 = \text{tnode } (A\Box++ x \ Z \ x_1 \ l \ m \ x_2)$$

$$\begin{aligned} A\Box-p+ &: \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} (Q : Process+ \infty \ c_1) \\ &\quad (Z : Process+ \infty \{lu\} \ c_2) \\ &\rightarrow (x : ChoiceSet \ c_0) \\ &\rightarrow (l : List (Label \ lu)) \\ &\rightarrow (m : Maybe ((ChoiceSet \ c_0 \ \uplus \ ChoiceSet \ c_1) \ \uplus \ ChoiceSet \ c_2)) \\ &\rightarrow (tr : Tr+ \ l \ m \ (fmap+ Ass\uplus r \ (addTimed\checkmark + (inj_1 \ x) \ (fmap+ inj_2 \ (Q \ \Box p++ Z))) \\ &\rightarrow Tr \ l \ m \ (\text{node } (addTimed\checkmark \ (inj_1 \ x) \ (fmap \ inj_2 \ Q) \ \Box p++ Z))) \end{aligned}$$

$$A\Box-p+ \ (\text{terminate } q) \ Z \ x \ l \ m \ tr = \text{tnode } (A\Box--+ Z \ q \ x \ l \ m \ tr) \ \text{--}$$

$$A\Box-p+ \ (\text{node } q) \ Z \ x \ l \ m \ tr = \text{tnode } (A\Box++ q \ Z \ x \ l \ m \ tr)$$

$$\begin{aligned} A\Box-\infty- &: \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} (Q : Process\infty \infty \{lu\} \ c_1) \\ &\rightarrow (m : Maybe maybeExtC) \\ &\rightarrow (l : List (Label \ lu)) \\ &\rightarrow (x : ChoiceSet \ c_0) \end{aligned}$$

```

    → (x1 : ChoiceSet c2)
    → (x3 : Tr∞ l m (fmap∞ Ass⊔r (addTimed✓∞ (inj1 x) (fmap∞ inj2 (addTimed✓∞ (i
    → Tr∞ l m (addTimed✓∞ (inj2 x1) (fmap∞ inj1 (addTimed✓∞ (inj1 x) (fmap∞ inj2 (i
A□-∞- Q m l x x1 tr = A□-p- (forcep Q) x1 x l m tr ---

```

```

A□∞-- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → ( x3 : Tr∞ l m (fmap∞ Ass⊔r (P □∞++ fmap+ unifyA⊔A (2-✓+ (inj1 x1) (inj2 x3
  → Tr∞ l m (addTimed✓∞ (inj2 x) (fmap∞ inj1 (addTimed✓∞ (inj2 x1) (fmap∞ inj1 (
A□∞-- P m l x x1 tr = A□p-- (forcep P) x1 x l m tr

```

```

A□∞--s : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (fmap∞ Ass⊔r (P □∞++ fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x3
  → Tr∞ l m (fmap∞ Ass⊔r (P □∞++ fmap+ unifyA⊔A (2-✓+ (inj1 x) (inj2 x1))))
A□∞--s P m l x x1 tr = A□p--s (forcep P) x1 x l m tr --

```

```

A□--∞s : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (x1 : ChoiceSet c1)
  → ( x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x3 : Tr∞ l m (fmap∞ Ass⊔r (addTimed✓∞ (inj1 x) (fmap∞ inj2 (addTimed✓∞ (i
  → Tr∞ l m (fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x)) □+∞+ Z)
A□--∞s Z q x l m tr = A□p--s (forcep Z) x q l m tr

```

```

A□--∞ : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (q : ChoiceSet c1)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)

```

$\rightarrow (x_2 : \text{Tr} \infty l m (\text{fmap} \infty \text{Ass} \uplus (\text{addTimed} \checkmark \infty (\text{inj}_1 x) (\text{fmap} \infty \text{inj}_2 (\text{addTimed} \checkmark \infty (\text{inj}_1 x) (\text{inj}_2 q)) \square + \infty + Z))$
 $\rightarrow \text{Tr} \infty l m (\text{fmap} + \text{unifyA} \uplus \text{A} (2\text{-}\checkmark + (\text{inj}_1 x) (\text{inj}_2 q)) \square + \infty + Z)$
 $\text{A} \square -- \infty Z q x l m tr = \text{A} \square -- p (\text{forcep } Z) x q l m tr$

$\text{A} \square -- p : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (Z : \text{Process} \infty c_2)$
 $\rightarrow (x : \text{ChoiceSet } c_0)$
 $\rightarrow (q : \text{ChoiceSet } c_1)$
 $\rightarrow (l : \text{List } (\text{Label } lu))$
 $\rightarrow (m : \text{Maybe } ((\text{ChoiceSet } c_0 \uplus \text{ChoiceSet } c_1) \uplus \text{ChoiceSet } c_2))$
 $\rightarrow (x_2 : \text{Tr } l m (\text{fmap } \text{Ass} \uplus (\text{addTimed} \checkmark (\text{inj}_1 x) (\text{fmap } \text{inj}_2 (\text{addTimed} \checkmark (\text{inj}_1 x) (\text{inj}_2 q)) \square + p + Z))$
 $\rightarrow \text{Tr } l m (\text{node } (\text{fmap} + \text{unifyA} \uplus \text{A} (2\text{-}\checkmark + (\text{inj}_1 x) (\text{inj}_2 q)) \square + p + Z))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . \text{nothing } (\text{tnode empty}) = \text{tnode empty}$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . (\text{Lab } (2\text{-}\checkmark + (\text{inj}_1 q) (\text{inj}_2 x)) - :: l_1) m (\text{tnode } (\text{extc } l_1 . m () tr))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q l m (\text{tnode } (\text{intc } . l . m () tr))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_1 x_1))) (\text{tnode } (\text{terc } (\text{inj}_1 \text{zero}))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_1 x_1)))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_1 x_1))) (\text{tnode } (\text{terc } (\text{inj}_1 (\text{suc } ()))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_1 x_1)))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_2 q))) (\text{tnode } (\text{terc } (\text{inj}_2 \text{zero}))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_2 q)))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_2 x)) (\text{tnode } (\text{terc } (\text{inj}_2 (\text{suc } \text{zero})))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_2 x)))$
 $\text{A} \square -- p (\text{terminate } x) x_1 q . [] . (\text{just } (\text{Ass} \uplus (\text{inj}_2 (\text{unifyA} \uplus \text{A} (\text{PT } (2\text{-}\checkmark + (\text{inj}_1 q) (\text{inj}_2 x)) (\text{suc } ())))$

$\text{A} \square -- p (\text{node } x) x_1 q l m (\text{tnode } tr) = \text{tnode } (\text{A} \square -- + x q x_1 l m tr)$

$\text{A} \square -- p_s : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (Z : \text{Process} \infty c_2)$
 $\rightarrow (x : \text{ChoiceSet } c_0)$
 $\rightarrow (q : \text{ChoiceSet } c_1)$
 $\rightarrow (l : \text{List } (\text{Label } lu))$
 $\rightarrow (m : \text{Maybe } ((\text{ChoiceSet } c_0 \uplus \text{ChoiceSet } c_1) \uplus \text{ChoiceSet } c_2))$
 $\rightarrow (x_2 : \text{Tr } l m (\text{fmap } \text{Ass} \uplus (\text{addTimed} \checkmark (\text{inj}_1 x) (\text{fmap } \text{inj}_2 (\text{addTimed} \checkmark (\text{inj}_1 x) (\text{inj}_2 q)) \square + p + Z))$
 $\rightarrow \text{Tr } l m (\text{node } (\text{fmap} + \text{unifyA} \uplus \text{A} (2\text{-}\checkmark + (\text{inj}_2 q) (\text{inj}_1 x)) \square + p + Z))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . \text{nothing } (\text{tnode empty}) = \text{tnode empty}$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . (\text{Lab } (2\text{-}\checkmark + (\text{inj}_1 q) (\text{inj}_2 x)) - :: l_1) m (\text{tnode } (\text{extc } l_1 . m () tr))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q l m (\text{tnode } (\text{intc } . l . m () tr))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_1 x_1))) (\text{tnode } (\text{terc } (\text{inj}_1 \text{zero}))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_1 x_1)))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_1 x_1))) (\text{tnode } (\text{terc } (\text{inj}_1 (\text{suc } ()))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_1 x_1)))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_1 (\text{inj}_2 q))) (\text{tnode } (\text{terc } (\text{inj}_2 \text{zero}))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_2 q)))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . (\text{just } (\text{inj}_2 x)) (\text{tnode } (\text{terc } (\text{inj}_2 (\text{suc } \text{zero})))) = \text{tnode } (\text{terc } (\text{inj}_1 (\text{inj}_2 x)))$
 $\text{A} \square -- p_s (\text{terminate } x) x_1 q . [] . (\text{just } (\text{Ass} \uplus (\text{inj}_2 (\text{unifyA} \uplus \text{A} (\text{PT } (2\text{-}\checkmark + (\text{inj}_1 q) (\text{inj}_2 x)) (\text{suc } (\text{inj}_2 x)) (\text{suc } (\text{suc } -)))))) (\text{tnode } (\text{terc } (\text{inj}_2 (\text{inj}_2 x)) (\text{suc } (\text{suc } -))))))$
 $\text{A} \square -- p_s (\text{node } x) x_1 q l m (\text{tnode } tr) = \text{A} \square -- +_s x x_1 q l m tr$

```

A□p-- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
  → (x1 : ChoiceSet c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x3 : Tr l m (node (fmap+ Ass⊔r (P □p++ fmap+ unifyA⊔A (2-✓+ (inj1 x1) (inj2 x2))
  → Tr l m (addTimed✓ (inj2 x) (fmap inj1 (addTimed✓ (inj2 x1) (fmap inj1 (P)))))))
A□p-- (terminate x) x1 x2 .[] .nothing (tnode empty) = tnode empty
A□p-- (terminate x) x1 x2 .(Lab (2-✓+ (inj1 x1) (inj2 x2)) - :: l1) m (tnode (extc l1 .m () tr))
A□p-- (terminate x) x1 x2 l m (tnode (intc .l .m () tr))
A□p-- (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj1 zero)))) = tnode (terc (inj2 zero))
A□p-- (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj1 (suc ())))))
A□p-- (terminate x) x1 x2 .[] .(just (inj1 (inj2 x1))) (tnode (terc (inj2 zero)))) = tnode (terc (inj2 zero))
A□p-- (terminate x) x1 x2 .[] .(just (inj2 x2)) (tnode (terc (inj2 (suc zero)))) = tnode (terc (inj1 zero))
A□p-- (terminate x) x1 x2 .[] .(just (Ass⊔r (inj2 (unifyA⊔A (PT (2-✓+ (inj1 x1) (inj2 x2)) (suc (suc -)
  (tnode (terc (inj2 (suc
A□p-- (node x) x1 x2 l m (tnode tr) = tnode (A□+-- x m l x2 x1 tr)

```

```

A□p--s : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
  → (x1 : ChoiceSet c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x3 : Tr l m (node (fmap+ Ass⊔r (P □p++ fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x2))
  → Tr l m (node (fmap+ Ass⊔r (P □p++ fmap+ unifyA⊔A (2-✓+ (inj1 x) (inj2 x1)))))))
A□p--s (terminate x) x1 x2 .[] .nothing (tnode empty) = tnode empty
A□p--s (terminate x) x1 x2 .(Lab (2-✓+ (inj2 x1) (inj1 x2)) - :: l1) m (tnode (extc l1 .m () tr))
A□p--s (terminate x) x1 x2 l m (tnode (intc .l .m () tr))
A□p--s (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj1 zero)))) = tnode (terc (inj1 zero))
A□p--s (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj1 (suc ())))))
A□p--s (terminate x) x1 x2 .[] .(just (inj2 x1)) (tnode (terc (inj2 zero)))) = tnode (terc (inj2 (suc zero))
A□p--s (terminate x) x1 x2 .[] .(just (inj1 (inj2 x2))) (tnode (terc (inj2 (suc zero)))) = tnode (terc (inj2 (suc zero))
A□p--s (terminate x) x1 x2 .[] .(just (Ass⊔r (inj2 (unifyA⊔A (PT (2-✓+ (inj2 x1) (inj1 x2)) (suc (suc -)
  (tnode (terc (inj2 (suc
A□p--s (node x) x1 x2 l m (tnode tr) = tnode (A□+--s x m l x2 x1 tr)

```

```

A□p- : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process ∞ c1)
  → (x1 : ChoiceSet c0)

```

[illegible][illegible][illegible]

```

A□+-- Q x1 x .[] .(just (inj2 x1)) (terc (inj2 (inj1 zero))) = terc (inj1 zero)
A□+-- Q x1 x .[] .(just (inj2 x1)) (terc (inj2 (inj1 (suc ())))))
A□+-- Q x1 x .[] .(just (inj1 (inj2 (PT Q y)))) (terc (inj2 (inj2 y))) = terc (inj2 (inj2 y))

```

```

A□+-- : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x2 : Tr+ l m (fmap+ Ass⊔r (P □++ fmap+ unifyA⊔A (2-✓+ (inj1 x1) (inj2 x))))))
  → Tr+ l m ((addTimed✓+ (inj2 x) (fmap+ inj1 (addTimed✓+ (inj2 x1) (fmap+ inj1 P))))
A□+-- P .nothing .[] x x1 empty = empty
A□+-- P m .(Lab P x2 :: l) x x1 (extc l .m (inj1 x2) tr) = let

```

```

x' : Tr∞ l m (fmap∞ Ass⊔r (fmap+ inj1 (fmap+ inj2 x)))
x' = tr

```

```

x1' : Tr∞ l m (fmap∞ (Ass⊔r (fmap+ inj1 (fmap+ inj2 x))))
x1' = lemFmap∞ inj1 Ass⊔r (PE P x2)

```

```

x2' : Tr∞ l m (fmap∞ inj1 (fmap+ inj2 x))
x2' = lemFmap∞R inj1 inj1 (PE P x2)

```

```

A□+-- P m .(Lab (2-✓+ (inj1 x1) (inj2 x)) - :: l1) x x1 (extc l1 .m (inj2 ())) tr
A□+-- P m l x x1 (intc .l .m (inj1 x2) tr) = intc l m x2 (A□∞-- (PI P x2) m l x x1 tr)
A□+-- P m l x x1 (intc .l .m (inj2 ())) tr
A□+-- P .(just (inj1 (inj1 (PT P x2)))) .[] x x1 (terc (inj1 x2)) = terc (inj2 (inj2 x2))
A□+-- P .(just (inj1 (inj2 x1))) .[] x x1 (terc (inj2 zero)) = terc (inj2 (inj1 zero))
A□+-- P .(just (inj2 x)) .[] x x1 (terc (inj2 (suc zero))) = terc (inj1 zero)
A□+-- P .(just (Ass⊔r (inj2 (unifyA⊔A (PT (2-✓+ (inj1 x1) (inj2 x)) (suc (suc -))))))) .[] x x1 (terc (

```

```

A□+--s : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x2 : Tr+ l m (fmap+ Ass⊔r (P □++ fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x))))))

```

$\rightarrow \text{Tr}+ \ l \ m \ (\text{fmap}+ \ \text{Ass}\oplus\text{r} \ (P \ \square++ \ \text{fmap}+ \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1)))$
 $\text{A}\square+--_s \ P \ .\text{nothing} \ .[] \ x \ x_1 \ \text{empty} = \text{empty}$
 $\text{A}\square+--_s \ P \ m \ .(\text{Lab} \ P \ x_2 :: l) \ x \ x_1 \ (\text{extc} \ l \ .m \ (\text{inj}_1 \ x_2) \ tr) = \text{let}$

$x' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ \text{Ass}\oplus\text{r} \ (P \ \square++ \ \text{fmap}\infty \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1))))$
 $x' = tr$

$x_1' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (P \ \square++ \ \text{fmap}\infty \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1))))$
 $x_1' = \text{lemFmap}\infty \ \text{inj}_1 \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (P \ \square++ \ \text{fmap}\infty \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1)))$

$x_2' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (P \ \square++ \ \text{fmap}\infty \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1))))$
 $x_2' = \text{lemFmap}\infty\text{R} \ \text{inj}_1 \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (P \ \square++ \ \text{fmap}\infty \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1)))$

$\text{A}\square+--_s \ P \ m \ .(\text{Lab} \ (2-\surd+ \ (\text{inj}_2 \ x_1) \ (\text{inj}_1 \ x)) \ - :: l_1) \ x \ x_1 \ (\text{extc} \ l_1 \ .m \ (\text{inj}_2 \ ()) \ tr)$
 $\text{A}\square+--_s \ P \ m \ l \ x \ x_1 \ (\text{intc} \ .l \ .m \ (\text{inj}_1 \ x_2) \ tr) = \text{intc} \ l \ m \ (\text{inj}_1 \ x_2) \ (\text{A}\square\infty--_s \ (\text{PI} \ P \ x_2) \ m \ l \ x \ x_1 \ tr)$
 $\text{A}\square+--_s \ P \ m \ l \ x \ x_1 \ (\text{intc} \ .l \ .m \ (\text{inj}_2 \ ()) \ tr)$
 $\text{A}\square+--_s \ P \ .(\text{just} \ (\text{inj}_1 \ (\text{inj}_1 \ (\text{PT} \ P \ x_2)))) \ .[] \ x \ x_1 \ (\text{terc} \ (\text{inj}_1 \ x_2)) = \text{terc} \ (\text{inj}_1 \ x_2)$
 $\text{A}\square+--_s \ P \ .(\text{just} \ (\text{inj}_2 \ x_1)) \ .[] \ x \ x_1 \ (\text{terc} \ (\text{inj}_2 \ \text{zero})) = \text{terc} \ (\text{inj}_2 \ (\text{suc} \ \text{zero}))$
 $\text{A}\square+--_s \ P \ .(\text{just} \ (\text{inj}_1 \ (\text{inj}_2 \ x))) \ .[] \ x \ x_1 \ (\text{terc} \ (\text{inj}_2 \ (\text{suc} \ \text{zero}))) = \text{terc} \ (\text{inj}_2 \ \text{zero})$
 $\text{A}\square+--_s \ P \ .(\text{just} \ (\text{Ass}\oplus\text{r} \ (\text{inj}_2 \ (\text{unifyA}\oplus\text{A} \ (\text{PT} \ (2-\surd+ \ (\text{inj}_2 \ x_1) \ (\text{inj}_1 \ x)) \ (\text{suc} \ (\text{suc} \ -)))))) \ .[] \ x \ x_1 \ (\text{terc} \ (\text{inj}_2 \ (\text{suc} \ (\text{suc} \ -)))) = \text{terc} \ (\text{inj}_2 \ (\text{suc} \ (\text{suc} \ -)))$

$\text{A}\square--+ \ : \ \{lu : \text{LUniv}\} \{c_0 \ c_1 \ c_2 : \text{Choice}\} (Z : \text{Process}+ \ \infty \ \{lu\} \ c_2)$
 $\rightarrow (q : \text{ChoiceSet} \ c_1)$
 $\rightarrow (x : \text{ChoiceSet} \ c_0)$
 $\rightarrow (l : \text{List} \ (\text{Label} \ lu))$
 $\rightarrow (m : \text{Maybe} \ ((\text{ChoiceSet} \ c_0 \ \uplus \ \text{ChoiceSet} \ c_1) \ \uplus \ \text{ChoiceSet} \ c_2))$
 $\rightarrow (tr : \text{Tr}+ \ l \ m \ (\text{fmap}+ \ \text{Ass}\oplus\text{r} \ (\text{addTimed}\surd+ \ (\text{inj}_1 \ x) \ (\text{fmap}+ \ \text{inj}_2 \ (\text{addTimed}\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ x_1))))$
 $\rightarrow \text{Tr}+ \ l \ m \ (\text{fmap}+ \ \text{unifyA}\oplus\text{A} \ (2-\surd+ \ (\text{inj}_1 \ x) \ (\text{inj}_2 \ q)) \ \square++ \ Z)$

$\text{A}\square--+ \ Z \ q \ x \ .[] \ .\text{nothing} \ \text{empty} = \text{empty}$

$\text{A}\square--+ \ Z \ q \ x \ .(\text{Lab} \ Z \ x_1 :: l) \ m \ (\text{extc} \ l \ .m \ x_1 \ x_2) = \text{let}$

$x' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ \text{Ass}\oplus\text{r} \ (\text{fmap}\infty \ \text{inj}_2 \ (\text{fmap}\infty \ \text{inj}_1 \ x)))$
 $x' = x_2$

$x_1' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_1) \ x))$
 $x_1' = \text{lemFmap}\infty \ \text{inj}_2 \ \text{Ass}\oplus\text{r} \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_1) \ x)$

$x_2' : \text{Tr}\infty \ l \ m \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2 \ o \ \text{inj}_1) \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_1) \ x))$
 $x_2' = \text{lemFmap}\infty \ \text{inj}_2 \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_2) \ (\text{fmap}\infty \ (\text{Ass}\oplus\text{r} \ o \ \text{inj}_1) \ x)$

$\text{in} \ \text{extc} \ l \ m \ (\text{inj}_2 \ x_1) \ x_2'$

```

A□--+ Z q x l m (intc .l .m x1 x2) = intc l m (inj2 x1) (A□--∞ (PI Z x1) q x l m x2)
A□--+ Z q x .[] .(just (inj1 (inj1 x))) (terc (inj1 zero)) = terc (inj1 zero)
A□--+ Z q x .[] .(just (inj1 (inj1 x))) (terc (inj1 (suc ())))
A□--+ Z q x .[] .(just (inj1 (inj2 q))) (terc (inj2 (inj1 zero))) = terc (inj1 (suc zero))
A□--+ Z q x .[] .(just (inj1 (inj2 q))) (terc (inj2 (inj1 (suc ())))))
A□--+ Z q x .[] .(just (inj2 (PT Z y))) (terc (inj2 (inj2 y))) = terc (inj2 y)

```

```

A□--+s      : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process+ ∞ {lu} c2)
              → ( x : ChoiceSet c0)
              → (x1 : ChoiceSet c1)
              → (l : List (Label lu))
              → (m : Maybe ((ChoiceSet c0 ⊔ ChoiceSet c1) ⊔ ChoiceSet c2))
              → ( x2 : Tr+ l m (fmap+ Ass⊔r (addTimed✓+ (inj1 x) (fmap+ inj2 (addTimed✓+ (i
              → Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x)) □++ Z))

```

```

A□--+s Z x x1 .[] .nothing empty = tnode empty

```

```

A□--+s Z x x1 .(Lab Z x2 :: l) m (extc l .m x2 x3) = let

```

```

  x' : Tr∞ l m (fmap∞ Ass⊔r (fmap∞ inj2 (fr
  x' = x3

```

```

  x1' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj2) (fmap∞
  x1' = lemFmap∞ inj2 Ass⊔r (fmap∞ inj2 (PI

```

```

  x2' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj2 ∘ inj2) (
  x2' = lemFmap∞ inj2 (Ass⊔r ∘ inj2) (PE Z x

```

```

  in tnode (extc l m (inj2 x2) x2')

```

```

A□--+s Z x x1 l m (intc .l .m x2 x3) = tnode (intc l m (inj2 x2) (A□--∞s (PI Z x2) x1 x l m x3))
A□--+s Z x x1 .[] .(just (inj1 (inj1 x))) (terc (inj1 zero)) = tnode (terc (inj1 (suc zero)))
A□--+s Z x x1 .[] .(just (inj1 (inj1 x))) (terc (inj1 (suc ())))
A□--+s Z x x1 .[] .(just (inj1 (inj2 x1))) (terc (inj2 (inj1 zero))) = tnode (terc (inj1 zero))
A□--+s Z x x1 .[] .(just (inj1 (inj2 x1))) (terc (inj2 (inj1 (suc ())))))
A□--+s Z x x1 .[] .(just (inj2 (PT Z y))) (terc (inj2 (inj2 y))) = tnode (terc (inj2 y))

```

mutual

```

A□+r : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0) (Q : Process+ ∞ {lu} c1) (Z : P
      → fmap+ Ass⊔r (P □++ (Q □++ Z)) ⊑+ ((P □++ Q) □++ Z)

```

```

A□+r P Q Z .[] .nothing empty = empty

```

$A\Box+r P Q Z .(\text{Lab } P x :: l) m (\text{extc } l .m (\text{inj}_1 (\text{inj}_1 x)) x_1) = \text{let}$

$x' : \text{Tr}_\infty l m (\text{fmap}_\infty \text{inj}_1 (\text{fmap}_\infty i$
 $x' = x_1$

$x_1' : \text{Tr}_\infty l m (\text{fmap}_\infty (\text{Ass} \oplus r$
 $x_1' = \text{lemFmap}_\infty \text{inj}_1 \text{inj}_1 (\text{PE } P x)$

$x_2' : \text{Tr}_\infty l m (\text{PE } (\text{fmap}_\infty$
 $x_2' = \text{lemFmap}_\infty R \text{inj}_1 \text{Ass} \oplus r (\text{PE } P$

$A\Box+r P Q Z .(\text{Lab } Q y :: l) m (\text{extc } l .m (\text{inj}_1 (\text{inj}_2 y)) x_1) = \text{let}$ $\text{in extc } l m (\text{inj}_1 x) x_2'$

$x' : \text{Tr}_\infty l m (\text{fmap}_\infty \text{inj}_1 (\text{fmap}_\infty i$
 $x' = x_1$

$x_1' : \text{Tr}_\infty l m (\text{fmap}_\infty (\text{Ass} \oplus r$
 $x_1' = \text{lemFmap}_\infty \text{inj}_2 \text{inj}_1 (\text{PE } Q y)$

$x_2' : \text{Tr}_\infty l m (\text{fmap}_\infty (\text{Ass} \oplus r$
 $x_2' = \text{lemFmap}_\infty R \text{inj}_1 (\text{Ass} \oplus r \circ \text{inj}_2$

$x_3' : \text{Tr}_\infty l m (\text{fmap}_\infty \text{Ass} \oplus r$
 $x_3' = \text{lemFmap}_\infty R \text{inj}_2 \text{Ass} \oplus r (\text{fmap}_\infty$

$\text{in extc } l m (\text{inj}_2 (\text{inj}_1 y)) x_2'$

$A\Box+r P Q Z .(\text{Lab } Z y :: l) m (\text{extc } l .m (\text{inj}_2 y) x_1) = \text{let}$

$x' : \text{Tr}_\infty l m (\text{fmap}_\infty \text{inj}_2$
 $x' = x_1$

$x_2' : \text{Tr}_\infty l m (\text{fmap}_\infty (\text{Ass} \oplus r \circ i$

$x_2' = \text{lemFmap}_\infty R \text{inj}_2 (\text{Ass} \oplus r \circ i$

$x_3' : \text{Tr}_\infty l m (\text{fmap}_\infty \text{Ass} \oplus r (\text{fmap}_\infty$

$x_3' = \text{lemFmap}_\infty R \text{inj}_2 \text{Ass} \oplus r (\text{fmap}_\infty$

$\text{in extc } l m (\text{inj}_2 (\text{inj}_2 y)) x_3'$

$A\Box+r P Q Z l m (\text{intc } l .m (\text{inj}_1 (\text{inj}_1 x)) x_1) = \text{intc } l m (\text{inj}_1 x) (A\Box_\infty++r (\text{PI } P x) Q Z l m$

$A\Box+r P Q Z l m (\text{intc } l .m (\text{inj}_1 (\text{inj}_2 y)) x_1) = \text{intc } l m (\text{inj}_2 (\text{inj}_1 y)) (A\Box_\infty++r P (\text{PI } Q y) l m$

$A\Box+r P Q Z l m (\text{intc } l .m (\text{inj}_2 y) x_1) = \text{intc } l m (\text{inj}_2 (\text{inj}_2 y)) (A\Box_\infty++r P Q (\text{PI } Z y) l m$

```

A□+r P Q Z .[] .(just (inj1 (inj1 (PT P x)))) (terc (inj1 (inj1 x))) = terc (inj1 x)
A□+r P Q Z .[] .(just (inj1 (inj2 (PT Q y)))) (terc (inj1 (inj2 y))) = terc (inj2 (inj1 y))
A□+r P Q Z .[] .(just (inj2 (PT Z y))) (terc (inj2 y)) = terc (inj2 (inj2 y))

```

```

A□∞++r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
          (Q : Process+ ∞ {lu} c1)
          (Z : Process+ ∞ {lu} c2)
  → Ref∞ (fmap∞ Ass⊔r ( P □∞++ (Q □++ Z))) (((P □∞++ Q) □∞++ Z))
A□∞++r P Q Z l m (tnode tr) = tnode (A□+ppr (forcep P) Q Z l m tr)

```

```

A□+∞+r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
          (Q : Process∞ ∞ {lu} c1)
          (Z : Process+ ∞ {lu} c2)
  → Ref∞ (fmap∞ Ass⊔r ( P □+∞+ (Q □∞++ Z))) (((P □+∞+ Q) □∞++ Z))
A□+∞+r P Q Z l m (tnode tr) = tnode (A□p+pr P (forcep Q) Z l m tr)

```

```

A□++∞r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
          (Q : Process+ ∞ {lu} c1)
          (Z : Process∞ ∞ {lu} c2)
  → Ref∞ (fmap∞ Ass⊔r ( P □+∞+ (Q □+∞+ Z))) (((P □++ Q) □+∞+ Z))
A□++∞r P Q Z l m (tnode tr) = tnode (A□pp+r P Q (forcep Z) l m tr)

```

```

A□+ppr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
          (Q : Process+ ∞ {lu} c1)
          (Z : Process+ ∞ {lu} c2)
  → Ref+ (fmap+ Ass⊔r ( P □p++ (Q □++ Z))) (((P □p++ Q) □++ Z))
A□+ppr (terminate x) Q Z l m tr = A□-+r Q Z x l m tr
A□+ppr (node x) Q Z l m tr = A□+r x Q Z l m tr

```

```

A□p+pr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
          (Q : Process ∞ c1)
          (Z : Process+ ∞ {lu} c2)
  → Ref+ (fmap+ Ass⊔r ( P □++ (Q □p++ Z))) (((P □+p+ Q) □++ Z))
A□p+pr P (terminate x) Z l m tr = A□+-+r P Z x l m tr
A□p+pr P (node x) Z l m tr = A□+r P x Z l m tr

```

$A\Box pp+r : \{lu : LUniv\}\{c_0\ c_1\ c_2 : Choice\}(P : Process+ \infty \{lu\}\ c_0)$
 $\quad (Q : Process+ \infty \{lu\}\ c_1)$
 $\quad (Z : Process \infty c_2)$
 $\rightarrow Ref+ (fmap+ Ass\Joinr (P \Box++ (Q \Box+p+ Z))) (((P \Box++ Q) \Box+p+ Z))$
 $A\Box pp+r\ P\ Q\ (terminate\ x)\ l\ m\ tr = A\Box++-r\ P\ Q\ m\ l\ x\ tr$
 $A\Box pp+r\ P\ Q\ (node\ x)\ l\ m\ tr = A\Box+r\ P\ Q\ x\ l\ m\ tr$

$A\Box++-r : \{lu : LUniv\}\{c_0\ c_1\ c_2 : Choice\}(P : Process+ \infty \{lu\}\ c_0)$
 $\quad (Q : Process+ \infty \{lu\}\ c_1)$
 $\rightarrow (m : Maybe\ maybeExtC)$
 $\rightarrow (l : List\ (Label\ lu))$
 $\rightarrow (x : ChoiceSet\ c_2)$
 $\rightarrow (tr : Tr+ l\ m\ (addTimed\checkmark+ (inj_2\ x)\ (fmap+ inj_1\ (P \Box++ Q))))$
 $\rightarrow Tr+ l\ m\ (fmap+ Ass\Joinr\ (P \Box++ addTimed\checkmark+ (inj_2\ x)\ (fmap+ inj_1\ Q)))$
 $A\Box++-r\ P\ Q\ .nothing\ .[]\ x\ empty = empty$
 $A\Box++-r\ P\ Q\ m\ .(Lab\ P\ x_1 :: l)\ x\ (extc\ l\ .m\ (inj_1\ x_1)\ x_2) = let$

$x' : Tr\infty\ l\ m\ (fmap\infty\ inj_1\ (fmap\infty\ inj_2\ x))$
 $x' = x_2$

$x'_3 : Tr\infty\ l\ m\ (fmap\infty\ inj_1\ (fmap\infty\ inj_2\ x))$
 $x'_3 = lemFmap\infty\ inj_1\ inj_2\ x$

$x_1' : Tr\infty\ l\ m\ (fmap\infty\ inj_1\ (fmap\infty\ inj_2\ x))$
 $x_1' = lemFmap\infty\ inj_1\ inj_2\ x$

$x_2' : Tr\infty\ l\ m\ (fmap\infty\ inj_1\ (fmap\infty\ inj_2\ x))$
 $x_2' = lemFmap\infty R\ inj_1\ inj_2\ x$

$in\ extc\ l\ m\ (inj_1\ x_1)\ x_2'$

$A\Box++-r\ P\ Q\ m\ .(Lab\ Q\ y :: l)\ x\ (extc\ l\ .m\ (inj_2\ y)\ x_2) = let$

$x' : Tr\infty\ l\ m\ (fmap\infty\ inj_1\ (fmap\infty\ inj_2\ y))$
 $x' = x_2$

$x_1' : Tr\infty\ l\ m\ (fmap\infty\ (Ass\Joinr \circ inj_2)\ y)$
 $x_1' = lemFmap\infty\ inj_2\ inj_1\ (PE\ Q\ y)$

$$x_2' : \text{Tr} \infty l m (\text{fmap} \infty (\text{Ass} \uplus r \circ \text{inj}_1) (\text{PE } Q y))$$

$$x_2' = \text{lemFmap} \infty R \text{inj}_1 (\text{Ass} \uplus r \circ \text{inj}_2) (\text{PE } Q y)$$

$$x_3' : \text{Tr} \infty l m (\text{fmap} \infty \text{Ass} \uplus r (\text{fmap} \infty \text{inj}_2 (\text{PE } Q y)))$$

$$x_3' = \text{lemFmap} \infty R \text{inj}_2 \text{Ass} \uplus r (\text{fmap} \infty \text{inj}_1 (\text{PE } Q y))$$

$$\text{in extc } l m (\text{inj}_2 y) x_3'$$

$$\text{A}\square++\text{-r } P Q m l x (\text{intc } l m (\text{inj}_1 x_1) x_2) = \text{intc } l m (\text{inj}_1 x_1) (\text{A}\square\infty++\text{-r } (\text{PI } P x_1) Q m l x x_2)$$

$$\text{A}\square++\text{-r } P Q m l x (\text{intc } l m (\text{inj}_2 y) x_2) = \text{intc } l m (\text{inj}_2 y) (\text{A}\square+\infty\text{-r } P (\text{PI } Q y) m l x x_2)$$

$$\text{A}\square++\text{-r } P Q .(\text{just } (\text{inj}_2 x)) .[] x (\text{terc } (\text{inj}_1 \text{zero})) = \text{terc } (\text{inj}_2 (\text{inj}_1 \text{zero}))$$

$$\text{A}\square++\text{-r } P Q .(\text{just } (\text{inj}_2 x)) .[] x (\text{terc } (\text{inj}_1 (\text{suc } ())))$$

$$\text{A}\square++\text{-r } P Q .(\text{just } (\text{inj}_1 (\text{inj}_1 (\text{PT } P x_1)))) .[] x (\text{terc } (\text{inj}_2 (\text{inj}_1 x_1))) = \text{terc } (\text{inj}_1 x_1)$$

$$\text{A}\square++\text{-r } P Q .(\text{just } (\text{inj}_1 (\text{inj}_2 (\text{PT } Q y)))) .[] x (\text{terc } (\text{inj}_2 (\text{inj}_2 y))) = \text{terc } (\text{inj}_2 (\text{inj}_2 y))$$

$$\text{A}\square++\text{-r } : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (Q : \text{Process} + \infty \{lu\} c_1)$$

$$(Z : \text{Process} + \infty \{lu\} c_2)$$

$$\rightarrow (x : \text{ChoiceSet } c_0)$$

$$\rightarrow (l : \text{List } (\text{Label } lu))$$

$$\rightarrow (m : \text{Maybe } ((\text{ChoiceSet } c_0 \uplus \text{ChoiceSet } c_1) \uplus \text{ChoiceSet } c_2))$$

$$\rightarrow (tr : \text{Tr} + l m (\text{addTimed} \checkmark + (\text{inj}_1 x) (\text{fmap} + \text{inj}_2 Q) \square++ Z))$$

$$\rightarrow \text{Tr} + l m (\text{fmap} + \text{Ass} \uplus r (\text{addTimed} \checkmark + (\text{inj}_1 x) (\text{fmap} + \text{inj}_2 (Q \square++ Z))))$$

$$\text{A}\square++\text{-r } Q Z x .[] .\text{nothing empty} = \text{empty}$$

$$\text{A}\square++\text{-r } Q Z x .(\text{Lab } Q x_1 :: l) m (\text{extc } l m (\text{inj}_1 x_1) x_2) = (\text{let}$$

$$x' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 (\text{PE } Q y))$$

$$x' = x_2$$

$$x_1' : \text{Tr} \infty l m (\text{fmap} \infty (\text{Ass} \uplus r \circ \text{inj}_1) (\text{PE } Q y))$$

$$x_1' = \text{lemFmap} \infty \text{inj}_2 \text{inj}_1 (\text{Ass} \uplus r \circ \text{inj}_1) (\text{PE } Q y)$$

$$x_2' : \text{Tr} \infty l m (\text{fmap} \infty (\text{Ass} \uplus r \circ \text{inj}_2) (\text{PE } Q y))$$

$$x_2' = \text{lemFmap} \infty R \text{inj}_1 (\text{Ass} \uplus r \circ \text{inj}_2) (\text{PE } Q y)$$

$$x_3' : \text{Tr} \infty l m (\text{fmap} \infty \text{Ass} \uplus r (\text{fmap} \infty \text{inj}_2 (\text{PE } Q y)))$$

$$x_3' = \text{lemFmap} \infty R \text{inj}_2 \text{Ass} \uplus r (\text{fmap} \infty \text{inj}_1 (\text{PE } Q y))$$

$$\text{in extc } l m (\text{inj}_1 x_1) x_3'$$

$$\text{A}\square++\text{-r } Q Z x .(\text{Lab } Z y :: l) m (\text{extc } l m (\text{inj}_2 y) x_2) =$$

$$\text{let}$$

$$x' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_2 (\text{PE } Z y))$$

$$x' = x_2$$

$$\begin{aligned}
 x_2' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, (\text{Ass} \oplus r \circ i) \\
 x_2' &= \text{lemFmap}^\infty R \, \text{inj}_2 \, (\text{Ass} \oplus r \circ i) \\
 x_3' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, \text{Ass} \oplus r \, (\text{fmap}^\infty \, i \\
 x_3' &= \text{lemFmap}^\infty R \, \text{inj}_2 \, \text{Ass} \oplus r \, (\text{fmap}^\infty \, i \\
 &\text{in } \text{extc} \, l \, m \, ((\text{inj}_2 \, y)) \, x_3'
 \end{aligned}$$

$$\begin{aligned}
 A \square \text{--} ++ r \, Q \, Z \, x \, l \, m \, (\text{intc} \, l \, m \, (\text{inj}_1 \, x_1) \, x_2) &= \text{intc} \, l \, m \, (\text{inj}_1 \, x_1) \, (A \square \text{--} \infty + r \, (\text{PI} \, Q \, x_1) \, Z \, x \, l \, m \, x_2) \\
 A \square \text{--} ++ r \, Q \, Z \, x \, l \, m \, (\text{intc} \, l \, m \, (\text{inj}_2 \, y) \, x_2) &= \text{intc} \, l \, m \, (\text{inj}_2 \, y) \, (A \square \text{--} \infty + r \, Q \, (\text{PI} \, Z \, y) \, x \, l \, m \, x_2) \\
 A \square \text{--} ++ r \, Q \, Z \, x \, [] \, (\text{just} \, (\text{inj}_1 \, (\text{inj}_1 \, x))) \, (\text{terc} \, (\text{inj}_1 \, (\text{inj}_1 \, \text{zero}))) &= \text{terc} \, (\text{inj}_1 \, \text{zero}) \\
 A \square \text{--} ++ r \, Q \, Z \, x \, [] \, (\text{just} \, (\text{inj}_1 \, (\text{inj}_1 \, x))) \, (\text{terc} \, (\text{inj}_1 \, (\text{inj}_1 \, (\text{suc} \, ()))) &= \text{terc} \, (\text{inj}_1 \, (\text{inj}_1 \, (\text{suc} \, ()))) \\
 A \square \text{--} ++ r \, Q \, Z \, x \, [] \, (\text{just} \, (\text{inj}_1 \, (\text{inj}_2 \, (\text{PT} \, Q \, y)))) \, (\text{terc} \, (\text{inj}_1 \, (\text{inj}_2 \, y))) &= \text{terc} \, (\text{inj}_2 \, (\text{inj}_1 \, y)) \\
 A \square \text{--} ++ r \, Q \, Z \, x \, [] \, (\text{just} \, (\text{inj}_2 \, (\text{PT} \, Z \, y))) \, (\text{terc} \, (\text{inj}_2 \, y)) &= \text{terc} \, (\text{inj}_2 \, (\text{inj}_2 \, y))
 \end{aligned}$$

$$\begin{aligned}
 A \square \text{--} ++ r &: \{lu : \text{LUniv}\} \{c_0 \, c_1 \, c_2 : \text{Choice}\} (P : \text{Process} + \infty \, \{lu\} \, c_0) \\
 &\quad (Z : \text{Process} + \infty \, \{lu\} \, c_2) \\
 &\rightarrow (x : \text{ChoiceSet} \, c_1) \\
 &\rightarrow (l : \text{List} \, (\text{Label} \, lu)) \\
 &\rightarrow (m : \text{Maybe} \, \text{maybeExtC}) \\
 &\rightarrow (tr : \text{Tr} + \, l \, m \, (\text{addTimed} \checkmark + \, (\text{inj}_2 \, x) \, (\text{fmap} + \, \text{inj}_1 \, P) \, \square ++ \, Z)) \\
 &\rightarrow \text{Tr} + \, l \, m \, (\text{fmap} + \, \text{Ass} \oplus r \, (P \, \square ++ \, \text{addTimed} \checkmark + \, (\text{inj}_1 \, x) \, (\text{fmap} + \, \text{inj}_2 \, Z)))
 \end{aligned}$$

$$A \square \text{--} ++ r \, P \, Z \, x \, [] \, \text{nothing} \, \text{empty} = \text{empty}$$

$$A \square \text{--} ++ r \, P \, Z \, x \, (\text{Lab} \, P \, x_1 :: l) \, m \, (\text{extc} \, l \, m \, (\text{inj}_1 \, x_1) \, x_2) = \text{let}$$

$$\begin{aligned}
 x' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, \text{inj}_1 \, P) \\
 x' &= x_2
 \end{aligned}$$

$$\begin{aligned}
 x_3' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, (\text{fmap}^\infty \, \text{inj}_1 \, P) \, \text{inj}_2 \, Z) \\
 x_3' &= \text{lemFmap}^\infty \, \text{inj}_1 \, \text{inj}_2 \, Z
 \end{aligned}$$

$$\begin{aligned}
 x_1' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, \text{inj}_1 \, P) \\
 x_1' &= \text{lemFmap}^\infty \, \text{inj}_1 \, P
 \end{aligned}$$

$$\begin{aligned}
 x_2' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, \text{inj}_1 \, P) \\
 x_2' &= \text{lemFmap}^\infty R \, \text{inj}_1 \, P
 \end{aligned}$$

$$\text{in } \text{extc} \, l \, m \, (\text{inj}_1 \, x_1) \, x_2'$$

$$A \square \text{--} ++ r \, P \, Z \, x \, (\text{Lab} \, Z \, y :: l) \, m \, (\text{extc} \, l \, m \, (\text{inj}_2 \, y) \, x_2) = \text{let}$$

$$\begin{aligned}
 x' &: \text{Tr}^\infty \, l \, m \, (\text{fmap}^\infty \, \text{inj}_2 \, Z) \\
 x' &= x_2
 \end{aligned}$$

$$A\Box-\infty+r \ Q \ Z \ x \ l \ m \ tr = A\Box-p+r \ (\text{forcep} \ Q) \ Z \ x \ l \ m \ tr$$

$$\begin{aligned} A\Box-\infty+r : \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} & (P : Process\infty \infty \{lu\} \ c_0) \\ & (Z : Process+ \infty \{lu\} \ c_2) \\ \rightarrow (x : ChoiceSet \ c_1) \\ \rightarrow (l : List \ (Label \ lu)) \\ \rightarrow (m : Maybe \ maybeExtC) \\ \rightarrow (x_2 : Tr\infty \ l \ m \ (addTimed\checkmark \infty \ (inj_2 \ x) \ (fmap\infty \ inj_1 \ (P)) \ \Box\infty++ \ Z)) \\ \rightarrow Tr\infty \ l \ m \ (fmap\infty \ Ass\oplus r \ (P \ \Box\infty++ \ addTimed\checkmark + \ (inj_1 \ x) \ (fmap+ \ inj_2 \ Z))) \\ A\Box-\infty+r \ P \ Z \ x \ l \ m \ tr = & \ \text{tnode} \ (A\Box-p+r \ ((\text{forcep} \ P)) \ Z \ x \ l \ m \ tr) \end{aligned}$$

$$\begin{aligned} A\Box+-\infty r : \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} & (P : Process+ \infty \{lu\} \ c_0) \\ & (Z : Process\infty \infty \{lu\} \ c_2) \\ \rightarrow (x : ChoiceSet \ c_1) \\ \rightarrow (l : List \ (Label \ lu)) \\ \rightarrow (m : Maybe \ maybeExtC) \\ \rightarrow (x_2 : Tr\infty \ l \ m \ (addTimed\checkmark + \ (inj_2 \ x) \ (fmap+ \ inj_1 \ P) \ \Box+\infty+ \ Z)) \\ \rightarrow Tr\infty \ l \ m \ (fmap\infty \ Ass\oplus r \ (P \ \Box+\infty+ \ addTimed\checkmark \infty \ (inj_1 \ x) \ (fmap\infty \ inj_2 \ (Z))) \\ A\Box+-\infty r \ P \ Z \ x \ l \ m \ tr = & A\Box+-pr \ P \ (\text{forcep} \ Z) \ x \ l \ m \ tr \end{aligned}$$

$$\begin{aligned} A\Box+-\infty r : \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} & (Q : Process+ \infty \{lu\} \ c_1) \\ & (Z : Process\infty \infty \{lu\} \ c_2) \\ \rightarrow (x : ChoiceSet \ c_0) \\ \rightarrow (l : List \ (Label \ lu)) \\ \rightarrow (m : Maybe \ maybeExtC) \\ \rightarrow (x_2 : Tr\infty \ l \ m \ (addTimed\checkmark + \ (inj_1 \ x) \ (fmap+ \ inj_2 \ Q) \ \Box+\infty+ \ Z)) \\ \rightarrow Tr\infty \ l \ m \ (fmap\infty \ Ass\oplus r \ (addTimed\checkmark \infty \ (inj_1 \ x) \ (fmap\infty \ inj_2 \ (Q \ \Box+\infty+ \ Z))) \\ A\Box+-\infty r \ Q \ Z \ x \ l \ m \ (\text{tnode} \ tr) = & \ \text{tnode} \ (A\Box+-pr \ Q \ (\text{forcep} \ Z) \ x \ l \ m \ tr) \end{aligned}$$

$$\begin{aligned} A\Box p+-r : \{lu : LUniv\} \{c_0 \ c_1 \ c_2 : Choice\} & (P : Process \ \infty \ c_0) \\ & (Q : Process+ \infty \{lu\} \ c_1) \\ \rightarrow (m : Maybe \ maybeExtC) \\ \rightarrow (l : List \ (Label \ lu)) \\ \rightarrow (x : ChoiceSet \ c_2) \\ \rightarrow (x_2 : Tr+ \ l \ m \ ((addTimed\checkmark + \ (inj_2 \ x) \ (fmap+ \ inj_1 \ (P \ \Box p++ \ Q)))) \\ \rightarrow Tr+ \ l \ m \ (fmap+ \ Ass\oplus r \ (P \ \Box p++ \ addTimed\checkmark + \ (inj_2 \ x) \ (fmap+ \ inj_1 \ Q))) \end{aligned}$$

```

A□p+-r (terminate x) Q m l x1 x2 = A□-+-r Q x1 x l m x2
A□p+-r (node x) Q m l x1 x2      = A□++-r x Q m l x1 x2

A□+p-r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Q : Process ∞ c1)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c2)
  → (x2 : Tr+ l m ((addTimed✓+ (inj2 x) (fmap+ inj1 (P □+p+ Q))))))
  → Tr+ l m (fmap+ Ass⊕r (P □+p+ addTimed✓ (inj2 x) (fmap inj1 Q))))
A□+p-r P (terminate x) m l x1 x2 = A□-+-r P m l x x1 x2
A□+p-r P (node x) m l x1 x2      = A□++-r P x m l x1 x2

```

```

A□+-pr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
      (Z : Process ∞ c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr l m (node (addTimed✓+ (inj2 x) (fmap+ inj1 P) □+p+ Z))))
  → Tr l m (node (fmap+ Ass⊕r (P □+p+ addTimed✓ (inj1 x) (fmap inj2 Z))))))
A□+-pr P (terminate x) x1 l m x2 = tnode (A□+--rr P m l x1 x x2)
A□+-pr P (node x) x1 l m (tnode tr) = tnode (A□+--rr P x x1 l m tr)

```

```

A□-+pr : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process+ ∞ {lu} c1)
      (Z : Process ∞ c2)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (tr : Tr+ l m ((addTimed✓+ (inj1 x) (fmap+ inj2 Q) □+p+ Z))))
  → Tr+ l m ((fmap+ Ass⊕r (addTimed✓+ (inj1 x) (fmap+ inj2 (Q □+p+ Z))))))
A□-+pr Q (terminate x) x1 l m tr = A□-+-r Q x x1 l m tr
A□-+pr Q (node x) x1 l m tr      = A□++-r Q x x1 l m tr

```

```

A□p+-r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
      (Z : Process+ ∞ {lu} c2)
  → (x : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr l m (node (addTimed✓ (inj2 x) (fmap inj1 P) □p++ Z))))

```

$x' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x' = x_3$

$x'_3 : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x'_3 = \text{lemFmap} \infty \text{inj}_1 \text{inj}_2$

$x_1' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x_1' = \text{lemFmap} \infty \text{inj}_1 \text{inj}_2$

$x_2' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x_2' = \text{lemFmap} \infty \text{R inj}_1 \text{inj}_2$

$\text{in extc } l m (\text{inj}_1 x_2) x_2'$
 $\text{A} \square + \text{--rr } P m l x x_1 (\text{tnode } (\text{intc } l m x_2 x_3)) = \text{intc } l m (\text{inj}_1 x_2) (\text{A} \square \text{--rr } (\text{PT } P))$
 $\text{A} \square + \text{--rr } P .(\text{just } (\text{inj}_2 x_1)) .[] x x_1 (\text{tnode } (\text{terc } (\text{inj}_1 \text{zero}))) = \text{terc } (\text{inj}_2 (\text{suc } \text{zero}))$
 $\text{A} \square + \text{--rr } P .(\text{just } (\text{inj}_2 x_1)) .[] x x_1 (\text{tnode } (\text{terc } (\text{inj}_1 (\text{suc } ())))))$
 $\text{A} \square + \text{--rr } P .(\text{just } (\text{inj}_1 (\text{inj}_2 x))) .[] x x_1 (\text{tnode } (\text{terc } (\text{inj}_2 (\text{inj}_1 \text{zero})))) = \text{terc } (\text{inj}_2 \text{zero})$
 $\text{A} \square + \text{--rr } P .(\text{just } (\text{inj}_1 (\text{inj}_2 x))) .[] x x_1 (\text{tnode } (\text{terc } (\text{inj}_2 (\text{inj}_1 (\text{suc } ())))))$
 $\text{A} \square + \text{--rr } P .(\text{just } (\text{inj}_1 (\text{inj}_1 (\text{PT } P y)))) .[] x x_1 (\text{tnode } (\text{terc } (\text{inj}_2 (\text{inj}_2 y)))) = \text{terc } (\text{inj}_1 y)$

$\text{A} \square + \text{--}_s r : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process} + \infty \{lu\} c_0)$
 $\rightarrow (m : \text{Maybe maybeExtC})$
 $\rightarrow (l : \text{List } (\text{Label } lu))$
 $\rightarrow (x : \text{ChoiceSet } c_1)$
 $\rightarrow (x_1 : \text{ChoiceSet } c_2)$
 $\rightarrow (x_2 : \text{Tr} + l m (\text{fmap} + \text{Ass} \oplus r (P \square ++ \text{fmap} + \text{unifyA} \oplus A (2\text{-}\checkmark + (\text{inj}_1 x) (\text{inj}_2 x))))$
 $\rightarrow \text{Tr} + l m (\text{fmap} + \text{Ass} \oplus r (P \square ++ \text{fmap} + \text{unifyA} \oplus A (2\text{-}\checkmark + (\text{inj}_2 x_1) (\text{inj}_1 x))))$

$\text{A} \square + \text{--}_s r P .\text{nothing} .[] x x_1 \text{empty} = \text{empty}$

$\text{A} \square + \text{--}_s r P m .(\text{Lab } P x_2 :: l) x x_1 (\text{extc } l m (\text{inj}_1 x_2) tr) = \text{let}$

$x' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x' = tr$

$x'_3 : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x'_3 = \text{lemFmap} \infty \text{inj}_1 \text{inj}_2$

$x_2' : \text{Tr} \infty l m (\text{fmap} \infty \text{inj}_1 \text{inj}_2)$
 $x_2' = \text{lemFmap} \infty \text{R inj}_1 \text{inj}_2$

$\text{in extc } l m (\text{inj}_1 x_2) x_2'$

```

A□+--s r P m .(Lab (2-✓+ (inj1 x) (inj2 x1)) - :: l1) x x1 (extc l1 .m (inj2 ()) tr)
A□+--s r P m l x x1 (intc .l .m (inj1 x2) tr) = intc l m (inj1 x2) (A□∞--s r (PI P x2) m l x x1 tr)
A□+--s r P m l x x1 (intc .l .m (inj2 ()) tr)
A□+--s r P .(just (inj1 (inj1 (PT P x2)))) .[] x x1 (terc (inj1 x2)) = terc (inj1 x2)
A□+--s r P .(just (inj1 (inj2 x))) .[] x x1 (terc (inj2 zero)) = terc (inj2 (suc zero))
A□+--s r P .(just (inj2 x1)) .[] x x1 (terc (inj2 (suc zero))) = terc (inj2 zero)
A□+--s r P .(just (Ass⊔r (inj2 (unifyA⊔A (PT (2-✓+ (inj1 x) (inj2 x1)) (suc (suc -))))))) .[] x x1 (terc

```

```

A□--+r      : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process+ ∞ {lu} c2)
              → (q : ChoiceSet c1)
              → (x : ChoiceSet c0)
              → (l : List (Label lu))
              → (m : Maybe ((ChoiceSet c0 ⊔ ChoiceSet c1) ⊔ ChoiceSet c2))
              → (tr : Tr+ l m (fmap+ unifyA⊔A (2-✓+ (inj1 x) (inj2 q)) □++ Z))
              → Tr+ l m (fmap+ Ass⊔r (addTimed✓+ (inj1 x) (fmap+ inj2 (addTimed✓+ (inj1 q) (f

```

```

A□--+r Z q x .[] .nothing empty = empty
A□--+r Z q x .(Lab (2-✓+ (inj1 x) (inj2 q)) - :: l1) m (extc l1 .m (inj1 ()) tr)
A□--+r Z q x .(Lab Z y :: l) m (extc l .m (inj2 y) tr) =
    let
        x' : Tr∞ l m (fmap∞ inj2 (PE Z y))
        x' = tr
        x2' : Tr∞ l m (fmap∞ (Ass⊔r ∘ inj2) (fmap∞ inj2 (PE Z y)))
        x2' = lemFmap∞R inj2 (Ass⊔r ∘ inj2) (PE Z y)
        x3' : Tr∞ l m (fmap∞ Ass⊔r (fmap∞ inj2 (PE Z y)))
        x3' = lemFmap∞R inj2 Ass⊔r (fmap∞ inj2 (PE Z y))
    in extc l m y x3'

```

```

A□--+r Z q x l m (intc .l .m (inj1 ()) tr)
A□--+r Z q x l m (intc .l .m (inj2 y) tr) = intc l m y (A□--∞r (PI Z y) q x l m tr)
A□--+r Z q x .[] .(just (inj1 (inj1 x))) (terc (inj1 zero)) = terc (inj1 zero)
A□--+r Z q x .[] .(just (inj1 (inj2 q))) (terc (inj1 (suc zero))) = terc (inj2 (inj1 zero))
A□--+r Z q x .[] .(just (inj1 (unifyA⊔A (PT (2-✓+ (inj1 x) (inj2 q)) (suc (suc -)))))) (terc (inj1 (suc (suc -)))) = terc (inj1 (suc (suc -)))
A□--+r Z q x .[] .(just (inj2 (PT Z y))) (terc (inj2 y)) = terc (inj2 (inj2 y))

```

```

A□--+rr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process+ ∞ {lu} c2)
          → (x1 : ChoiceSet c1)
          → (x : ChoiceSet c0)
          → (l : List (Label lu))
          → (m : Maybe ((ChoiceSet c0 ⊔ ChoiceSet c1) ⊔ ChoiceSet c2))
          → (tr : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj2 x1) (inj1 x)) □++ Z)))

```

```

→ Tr+ l m (fmap+ Ass⊔r (addTimed✓+ (inj₁ x) (fmap+ inj₂ (addTimed✓+ (i
A□--+rr Z x₁ x .[] .nothing (tnode empty) = empty
A□--+rr Z x₁ x .(Lab (2-✓+ (inj₂ x₁) (inj₁ x)) - :: l₁) m (tnode (extc l₁ .m (inj₁ ()) tr))
A□--+rr Z x₁ x .(Lab Z y :: l) m (tnode (extc l .m (inj₂ y) tr)) = let
    x' : Tr∞ l m (fmap∞ inj₂ (PE Z y))
    x' = tr

x₂' : Tr∞ l m (fmap∞ (Ass⊔r ∘ i
x₂' = lemFmap∞R inj₂ (Ass⊔r ∘ i
x₃' : Tr∞ l m (fmap∞ Ass⊔r (fm
x₃' = lemFmap∞R inj₂ Ass⊔r (fm
    in extc l m

A□--+rr Z x₁ x l m (tnode (intc .l .m (inj₁ ()) tr))
A□--+rr Z x₁ x l m (tnode (intc .l .m (inj₂ y) tr)) = intc l m y (A□--∞rrr (PI Z y) x x l m)
A□--+rr Z x₁ x .[] .(just (inj₁ (inj₂ x₁))) (tnode (terc (inj₁ zero)))) = terc (inj₂ (inj₁ zero))
A□--+rr Z x₁ x .[] .(just (inj₁ (inj₁ x))) (tnode (terc (inj₁ (suc zero)))) = terc (inj₁ zero)
A□--+rr Z x₁ x .[] .(just (inj₁ (unifyA⊔A (PT (2-✓+ (inj₂ x₁) (inj₁ x)) (suc (suc -)))))) (tnode (terc (inj₁ (suc zero)))) = terc (inj₁ (suc zero))
A□--+rr Z x₁ x .[] .(just (inj₂ (PT Z y))) (tnode (terc (inj₂ y))) = terc (inj₂ (inj₂ y))

A□--+rrrr : {lu : LUniv}{c₀ c₁ c₂ : Choice}(Z : Process+ ∞ {lu} c₂)
→ (q : ChoiceSet c₁)
→ (x : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe ((ChoiceSet c₀ ⊔ ChoiceSet c₁) ⊔ ChoiceSet c₂))
→ (tr : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj₁ x) (inj₂ q)) □++ Z)))
→ Tr+ l m ((fmap+ Ass⊔r (addTimed✓+ (inj₁ x) (fmap+ inj₂ (addTimed✓+ (i
A□--+rrrr Z q x .[] .nothing (tnode empty) = empty
A□--+rrrr Z q x .(Lab (2-✓+ (inj₁ x) (inj₂ q)) - :: l₁) m (tnode (extc l₁ .m (inj₁ ()) tr))
A□--+rrrr Z q x .(Lab Z y :: l) m (tnode (extc l .m (inj₂ y) tr)) = let
    x' : Tr∞ l m (fmap∞ inj₂ (PE Z y))
    x' = tr

x₂' : Tr∞ l m (fmap∞ (Ass⊔r ∘ i
x₂' = lemFmap∞R inj₂ (Ass⊔r ∘ i
x₃' : Tr∞ l m (fmap∞ Ass⊔r (fm
x₃' = lemFmap∞R inj₂ Ass⊔r (fm
    in extc l m      y x₃'

A□--+rrrr Z q x l m (tnode (intc .l .m (inj₁ ()) tr))
A□--+rrrr Z q x l m (tnode (intc .l .m (inj₂ y) tr)) = intc l m y (A□--∞rrrr (PI Z y) x q l m)
A□--+rrrr Z q x .[] .(just (inj₁ (inj₁ x))) (tnode (terc (inj₁ zero)))) = terc (inj₁ zero)

```

```

A□--+rrrr Z q x .[] .(just (inj1 (inj2 q))) (tnode (terc (inj1 (suc zero)))) = terc (inj2 (inj1 zero))
A□--+rrrr Z q x .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj1 x) (inj2 q)) (suc (suc -)))))) (tnode (terc (
A□--+rrrr Z q x .[] .(just (inj2 (PT Z y))) (tnode (terc (inj2 y))) = terc (inj2 (inj2 y))

```

```

A□--+srr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process+ ∞ {lu} c2)
  → ( x : ChoiceSet c0)
  → (x1 : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe ((ChoiceSet c0 ⊕ ChoiceSet c1) ⊕ ChoiceSet c2))
  → ( x2 : Tr+ l m ((fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)) □++ Z)))
  → Tr l m (node (fmap+ Ass⊕r (addTimed✓+ (inj1 x) (fmap+ inj2 (addTimed✓+ (inj1
A□--+srr Z x x1 .[] .nothing empty = tnode empty
A□--+srr Z x x1 .(Lab (2-✓+ (inj2 x1) (inj1 x)) - :: l1) m (extc l1 .m (inj1 ()) tr)
A□--+srr Z x x1 .(Lab Z y :: l) m (extc l .m (inj2 y) tr) = let
  x' : Tr∞ l m (fmap∞ inj2 (PE Z y))
  x' = tr

```

```

  x2' : Tr∞ l m (fmap∞ (Ass⊕r ∘ inj2) (fmap∞ inj1 (PE Z y)))
  x2' = lemFmap∞R inj2 (Ass⊕r ∘ inj2) (PE Z y)
  x3' : Tr∞ l m (fmap∞ Ass⊕r (fmap∞ inj2 (PE Z y)))
  x3' = lemFmap∞R inj2 Ass⊕r (fmap∞ inj1 (PE Z y))
  in tnode (extc l m y x3')

```

```

A□--+srr Z x x1 l m (intc .l .m (inj1 ()) tr)
A□--+srr Z x x1 l m (intc .l .m (inj2 y) tr) = tnode (intc l m y (A□--∞Lrrrrrrrr (PI Z y) x x1 l m tr))
A□--+srr Z x x1 .[] .(just (inj1 (inj2 x1))) (terc (inj1 zero)) = tnode (terc (inj2 (inj1 zero)))
A□--+srr Z x x1 .[] .(just (inj1 (inj1 x))) (terc (inj1 (suc zero))) = tnode (terc (inj1 zero))
A□--+srr Z x x1 .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj2 x1) (inj1 x)) (suc (suc -)))))) (terc (inj1 (suc zero)))
A□--+srr Z x x1 .[] .(just (inj2 (PT Z y))) (terc (inj2 y)) = tnode (terc (inj2 (inj2 y)))

```

```

A□-∞-r : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process∞ ∞ {lu} c1)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (addTimed✓∞ (inj2 x1) (fmap∞ inj1 (addTimed✓∞ (inj1 x) (fmap∞ inj2 (PE Z y))))))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x) (fmap∞ inj2 (addTimed✓∞ (inj1 x) (fmap∞ inj2 (PE Z y))))))
A□-∞-r Q m l x x1 tr = A□-p-r (forcep Q) x1 x l m tr

```

```

A□∞--r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (addTimed✓∞ (inj2 x) (fmap∞ inj1 (addTimed✓∞ (inj2 x)
  → Tr∞ l m (fmap∞ Ass⊕r (P □∞++ fmap+ unifyA⊕A (2-✓+ (inj1 x1) (inj2 x)
A□∞--r P m l x x1 tr = A□p--rr (forcep P) x1 x l m tr

```

```

A□∞--rr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (addTimed✓∞ (inj2 x1) (fmap∞ inj1 (addTimed✓∞ (inj2 x)
  → Tr∞ l m (fmap∞ Ass⊕r (P □∞++ fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)
A□∞--rr P m l x x1 tr = A□p--rrr (forcep P) x1 x l m tr --

```

```

A□∞--sr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process∞ ∞ {lu} c0)
  → (m : Maybe maybeExtC)
  → (l : List (Label lu))
  → (x : ChoiceSet c1)
  → (x1 : ChoiceSet c2)
  → (x3 : Tr∞ l m (fmap∞ Ass⊕r (P □∞++ fmap+ unifyA⊕A (2-✓+ (inj1 x) (inj2 x)
  → Tr∞ l m (fmap∞ Ass⊕r (P □∞++ fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)
A□∞--sr P m l x x1 tr = A□p--sr (forcep P) x1 x l m tr

```

```

A□--∞r : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (q : ChoiceSet c1)
  → (x : ChoiceSet c0)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr∞ l m (fmap+ unifyA⊕A (2-✓+ (inj1 x) (inj2 q)) □+∞+ Z))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x) (fmap∞ inj2 (addTimed✓∞
A□--∞r Z q x l m tr = A□p--srrrr (forcep Z) x q l m tr --

```

```

A□--∞rrr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr∞ l m (fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)) □+∞+ Z))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x)(fmap∞ inj2 (addTimed✓∞ (inj1 x1))
A□--∞rrr Z q x l m tr = A□p--srrrr (forcep Z) x q l m tr --

```

```

A□--∞rrrr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr∞ l m (fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)) □+∞+ Z))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x)(fmap∞ inj2 (addTimed✓∞ (inj1 x1))
A□--∞rrrr Z q x l m tr = A□p--srrrr (forcep Z) x q l m tr --

```

```

A□--∞rrrrr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (q : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr∞ l m (fmap+ unifyA⊕A (2-✓+ (inj1 x) (inj2 q)) □+∞+ Z))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x)(fmap∞ inj2 (addTimed✓∞ (inj1 q))
A□--∞rrrrr Z q x l m tr = A□p--srrrrr (forcep Z) x q l m tr --

```

```

A□--∞£rrrrrrr : {lu : LUniv}{c0 c1 c2 : Choice}(Z : Process∞ ∞ {lu} c2)
  → (x : ChoiceSet c0)
  → (x1 : ChoiceSet c1)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → (x2 : Tr∞ l m (fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1 x)) □+∞+ Z))
  → Tr∞ l m (fmap∞ Ass⊕r (addTimed✓∞ (inj1 x)(fmap∞ inj2 (addTimed✓∞ (inj1 x1))
A□--∞£rrrrrrr Z q x l m tr = A□p--srrrrrrr (forcep Z) x q l m tr --

```

```

A□p--rr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)

```

```

→ (x1 : ChoiceSet c2)
→ (x : ChoiceSet c1)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x3 : Tr l m ((addTimed✓ (inj2 x) (fmap inj1 (addTimed✓ (inj2 x1) (fmap inj1
→ Tr l m (node (fmap+ Ass⊕r (P □p++ fmap+ unifyA⊕A (2-✓+ (inj1 x1) (inj1
A□p--rr (terminate x) x1 x2 .[] .nothing (tnode empty) = tnode empty
A□p--rr (terminate x) x1 x2 .(Lab (2-✓+ (inj2 x1) (inj1 x)) - :: l1) m (tnode (extc l1 .m () tr))
A□p--rr (terminate x) x1 x2 l m (tnode (intc .l .m () tr))
A□p--rr (terminate x) x1 x2 .[] .(just (inj2 x2)) (tnode (terc (inj1 zero))) = tnode (terc (inj2 (s
A□p--rr (terminate x) x1 x2 .[] .(just (inj2 x2)) (tnode (terc (inj1 (suc ())))))
A□p--rr (terminate x) x1 x2 .[] .(just (inj1 (inj2 x1))) (tnode (terc (inj2 zero))) = tnode (terc (
A□p--rr (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj2 (suc zero)))) = tnode (t
A□p--rr (terminate x) x1 x2 .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj2 x1) (inj1 x)) (suc (suc
(tnode (terc (
A□p--rr (node x) x1 x2 l m x3 = tnode (A□+--rr x m l x1 x2 x3)

```

```

A□p--rrr : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
→ (x1 : ChoiceSet c2)
→ (x : ChoiceSet c1)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x3 : Tr l m ((addTimed✓ (inj2 x1)(fmap inj1 (addTimed✓ (inj2 x) (fmap inj1
→ Tr l m (node (fmap+ Ass⊕r (P □p++ fmap+ unifyA⊕A (2-✓+ (inj2 x1) (inj1
A□p--rrr (terminate x) x1 x2 .[] .nothing (tnode empty) = tnode empty
A□p--rrr (terminate x) x1 x2 .(Lab (2-✓+ (inj2 x2) (inj1 x)) - :: l1) m (tnode (extc l1 .m () tr))
A□p--rrr (terminate x) x1 x2 l m (tnode (intc .l .m () tr))
A□p--rrr (terminate x) x1 x2 .[] .(just (inj2 x1)) (tnode (terc (inj1 zero))) = tnode (terc (inj2 (s
A□p--rrr (terminate x) x1 x2 .[] .(just (inj2 x1)) (tnode (terc (inj1 (suc ())))))
A□p--rrr (terminate x) x1 x2 .[] .(just (inj1 (inj2 x2))) (tnode (terc (inj2 zero))) = tnode (terc (
A□p--rrr (terminate x) x1 x2 .[] .(just (inj1 (inj1 x))) (tnode (terc (inj2 (suc zero)))) = tnode (
A□p--rrr (terminate x) x1 x2 .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj2 x2) (inj1 x)) (suc (suc
(tnode (terc (
A□p--rrr (node x) x1 x2 l m (tnode x3) = tnode (A□+--r x m l x2 x1 x3)

```

```

A□p--s r : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ c0)
→ (x1 : ChoiceSet c2)

```

```

→ (x : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x₃ : Tr l m (node (fmap+ Ass⊔r (P □p++ fmap+ unifyA⊔A (2-✓+ (inj₁ x) (inj₂ x₁)
→ Tr l m (node (fmap+ Ass⊔r (P □p++ fmap+ unifyA⊔A (2-✓+ (inj₂ x₁) (inj₁ x))))))
A□p--s r (terminate x) x₁ x₂ .[] .nothing (tnode empty) = tnode empty
A□p--s r (terminate x) x₁ x₂ .(Lab (2-✓+ (inj₁ x₂) (inj₂ x₁)) - :: l₁) m (tnode (extc l₁ .m () tr))
A□p--s r (terminate x) x₁ x₂ l m (tnode (intc .l .m () tr))
A□p--s r (terminate x) x₁ x₂ .[] .(just (inj₁ (inj₁ x))) (tnode (terc (inj₁ zero))) = tnode (terc (inj₁ zero))
A□p--s r (terminate x) x₁ x₂ .[] .(just (inj₁ (inj₁ x))) (tnode (terc (inj₁ (suc ())))))
A□p--s r (terminate x) x₁ x₂ .[] .(just (inj₁ (inj₂ x₂))) (tnode (terc (inj₂ zero))) = tnode (terc (inj₂ (suc
A□p--s r (terminate x) x₁ x₂ .[] .(just (inj₂ x₁)) (tnode (terc (inj₂ (suc zero)))) = tnode (terc (inj₂ zero
A□p--s r (terminate x) x₁ x₂ .[] .(just (Ass⊔r (inj₂ (unifyA⊔A (PT (2-✓+ (inj₁ x₂) (inj₂ x₁)) (suc (suc
(tnode (terc (inj₂ (suc
A□p--s r (node x) x₁ x₂ l m (tnode tr) = tnode (A□+--s r x m l x₂ x₁ tr)

```

```

A□p--srrrr : {lu : LUniv}{c₀ c₁ c₂ : Choice} (Z : Process ∞ c₂)
→ (x : ChoiceSet c₀)
→ (q : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x₃ : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj₁ x) (inj₂ q)) □+p+ Z)))
→ Tr l m (fmap Ass⊔r (addTimed✓ (inj₁ x)(fmap inj₂ (addTimed✓ (inj₁ q) (fmap inj₂ Z))))
A□p--srrrr (terminate x) x₁ q .[] .nothing (tnode empty) = tnode empty
A□p--srrrr (terminate x) x₁ q .(Lab (2-✓+ (inj₁ x₁) (inj₂ q)) - :: l₁) m (tnode (extc l₁ .m () tr))
A□p--srrrr (terminate x) x₁ q l m (tnode (intc .l .m () tr))
A□p--srrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ zero))) = tnode (terc (inj₂ (suc zero))
A□p--srrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ (suc ())))))
A□p--srrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₁ x₁))) (tnode (terc (inj₂ zero))) = tnode (terc (inj₁ zer
A□p--srrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₂ q))) (tnode (terc (inj₂ (suc zero)))) = tnode (terc (in
A□p--srrrr (terminate x) x₁ q .[] .(just (inj₁ (unifyA⊔A (PT (2-✓+ (inj₁ x₁) (inj₂ q)) (suc (suc _))))))
(tnode (terc (inj₂ (suc
A□p--srrrr (node x) x₁ q l m (tnode tr) = tnode (A□--r x q x₁ l m tr)

```

```

A□p--srrrr : {lu : LUniv}{c₀ c₁ c₂ : Choice} (Z : Process ∞ c₂)
→ (x : ChoiceSet c₀)
→ (q : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x₃ : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj₂ x) (inj₁ q)) □+p+ Z)))

```

```

→ Tr l m (fmap Ass⊔r (addTimed✓ (inj₁ q)(fmap inj₂ (addTimed✓ (inj₁ x) (fmap
A□p--srrrrr (terminate x) x₁ q .[] .nothing (tnode empty) = tnode empty
A□p--srrrrr (terminate x) x₁ q .(Lab (2-✓+ (inj₂ x₁) (inj₁ q)) - :: l₁) m (tnode (extc l₁ .m ()
A□p--srrrrr (terminate x) x₁ q l m (tnode (intc .l .m () tr))
A□p--srrrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ x₂)))) = tnode (terc (inj₂ (suc
A□p--srrrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₂ x₁))) (tnode (terc (inj₂ zero)))) = tnode (terc
A□p--srrrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₁ q))) (tnode (terc (inj₂ (suc zero)))) = tnode
A□p--srrrrr (terminate x) x₁ q .[] .(just (inj₁ (unifyA⊔A (PT (2-✓+ (inj₂ x₁) (inj₁ q)) (suc (suc
(tnode (terc
A□p--srrrrr (node x) x₁ q l m x₃ = tnode (A□--+rr x x₁ q l m x₃)

```

```

A□p--srrrrrr : {lu : LUniv}{c₀ c₁ c₂ : Choice} (Z : Process ∞ c₂)
→ (x : ChoiceSet c₀)
→ (q : ChoiceSet c₁)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x₃ : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj₁ q) (inj₂ x)) □+p+ Z)))
→ Tr l m (fmap Ass⊔r (addTimed✓ (inj₁ q)(fmap inj₂ (addTimed✓ (inj₁ x) (fmap
A□p--srrrrrr (terminate x) x₁ q .[] .nothing (tnode empty) = tnode empty
A□p--srrrrrr (terminate x) x₁ q .(Lab (2-✓+ (inj₁ q) (inj₂ x₁)) - :: l₁) m (tnode (extc l₁ .m ()
A□p--srrrrrr (terminate x) x₁ q l m (tnode (intc .l .m () tr))
A□p--srrrrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ zero)))) = tnode (terc (inj₂
A□p--srrrrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ (suc ())))))
A□p--srrrrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₁ q))) (tnode (terc (inj₂ zero)))) = tnode (terc
A□p--srrrrrr (terminate x) x₁ q .[] .(just (inj₁ (inj₂ x₁))) (tnode (terc (inj₂ (suc zero)))) = tnode
A□p--srrrrrr (terminate x) x₁ q .[] .(just (inj₁ (unifyA⊔A (PT (2-✓+ (inj₁ q) (inj₂ x₁))
(suc (suc -)))))) (tnode (
A□p--srrrrrr (node x) x₁ q l m x₃ = tnode (A□--+rrrr x x₁ q l m x₃)

```

```

A□p--srrrrrrrr : {lu : LUniv}{c₀ c₁ c₂ : Choice} (Z : Process ∞ c₂)
→ (x : ChoiceSet c₁)
→ (q : ChoiceSet c₀)
→ (l : List (Label lu))
→ (m : Maybe maybeExtC)
→ (x₃ : Tr l m (node (fmap+ unifyA⊔A (2-✓+ (inj₂ x) (inj₁ q)) □+p+ Z)))
→ Tr l m (fmap Ass⊔r (addTimed✓ (inj₁ q)(fmap inj₂ (addTimed✓ (inj₁ x) (fmap
A□p--srrrrrrrr (terminate x) x₁ q .[] .nothing (tnode empty) = tnode empty
A□p--srrrrrrrr (terminate x) x₁ q .(Lab (2-✓+ (inj₂ x₁) (inj₁ q)) - :: l₁) m (tnode (extc l₁ .m ()
A□p--srrrrrrrr (terminate x) x₁ q l m (tnode (intc .l .m () tr))
A□p--srrrrrrrr (terminate x) x₁ q .[] .(just (inj₂ x)) (tnode (terc (inj₁ zero)))) = tnode (terc (inj

```

```

A□p--srrrrrrrr (terminate x) x1 q .[] .(just (inj2 x)) (tnode (terc (inj1 (suc ())))))
A□p--srrrrrrrr (terminate x) x1 q .[] .(just (inj1 (inj2 x1))) (tnode (terc (inj2 zero))) = tnode (terc (inj2 zero))
A□p--srrrrrrrr (terminate x) x1 q .[] .(just (inj1 (inj1 q))) (tnode (terc (inj2 (suc zero)))) = tnode (terc (inj2 (suc zero)))
A□p--srrrrrrrr (terminate x) x1 q .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj2 x1) (inj1 q))
                                                                    (suc (suc _)))))) (tnode (terc (inj2 (suc (suc _))))))
A□p--srrrrrrrr (node x) x1 q l m (tnode x3) = A□--srr x q x1 l m x3

A□-p-r : {lu : LUniv}{c0 c1 c2 : Choice}(Q : Process ∞ c1)
  → (x1 : ChoiceSet c0)
  → (x : ChoiceSet c2)
  → (l : List (Label lu))
  → (m : Maybe maybeExtC)
  → ( x3 : Tr l m (addTimed✓ (inj2 x1) (fmap inj1 (addTimed✓ (inj1 x) (fmap inj2 ( Q ))
                                                                    (suc (suc _))))))
  → Tr l m (fmap Ass⊕r (addTimed✓ (inj1 x) (fmap inj2 (addTimed✓ (inj2 x1) (fmap inj1
                                                                    ( Q ))
                                                                    (suc (suc _))))))
A□-p-r (terminate x) x1 x2 .[] .nothing (tnode empty) = tnode empty
A□-p-r (terminate x) x1 x2 .(Lab (2-✓+ (inj1 x2) (inj2 x)) - :: l1) m (tnode (extc l1 .m () tr))
A□-p-r (terminate x) x1 x2 l m (tnode (intc .l .m () tr))
A□-p-r (terminate x) x1 x2 .[] .(just (inj2 x1)) (tnode (terc (inj1 zero))) = tnode (terc (inj2 zero))
A□-p-r (terminate x) x1 x2 .[] .(just (inj2 x1)) (tnode (terc (inj1 (suc x3)))) = tnode (terc (inj2 zero))
A□-p-r (terminate x) x1 x2 .[] .(just (inj1 (inj1 x2))) (tnode (terc (inj2 zero))) = tnode (terc (inj1 zero))
A□-p-r (terminate x) x1 x2 .[] .(just (inj1 (inj2 x))) (tnode (terc (inj2 (suc zero)))) = tnode (terc (inj2 (suc zero)))
A□-p-r (terminate x) x1 x2 .[] .(just (inj1 (unifyA⊕A (PT (2-✓+ (inj1 x2) (inj2 x)) (suc (suc _))))))
                                                                    (tnode (terc (inj2 (suc (suc _))))))
A□-p-r (node x) x1 x2 l m (tnode x3) = tnode (A□-+s-r x x1 x2 l m x3)

```

```
--@BEGIN@AssExEqDef
```

```

≡A□+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : Process+ ∞ {lu} c1)
  (Z : Process+ ∞ {lu} c2)
  → ((P □++ Q) □++ Z) ≡+ fmap+ Ass⊕r (P □++ (Q □++ Z))

```

```
--@END
```

```
--@BEGIN@AssExEqDefProof

≡A□+ P Q Z = (A□+ P Q Z) , A□+r P Q Z

--@END
```

A.72 proofBisimForInterleaving.agda

```
--@PREFIX@proofBisimForInterleaving

module proofBisimForInterleaving where

open import process
open import choiceSetU
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Sum
open import renamingResult
open import lemFmap
open import auxData
open import interleave
open import RefWithoutSize
open import bisimilarity
open import bisimSImpliesBisimw
open import bisimilarityProofs
open import bisimImpliesTraceEquiv
open import bisimLemFmap
open import bisimImpliesFDI
open import fdiRefusal
open import bisimImpliesBisim
open import labelUniv
open import bisimImpliesFDIPartTwo
open import bisimImpliesTraceEquiv
open import traceEquivalence
open import bisimwImpliesStableFailuresEquivalence

mutual
--@BEGIN@SymIntBisim
```

```

C|||+  : {lu : LUniv} {c0 c1 : Choice}
        (P : Process+ ∞ {lu} c0)
        (Q : Process+ ∞ {lu} c1)
        → Bisims+ (P |||++ Q) (fmap+ swap× (Q |||++ P))

--@END

--@BEGIN@SymIntBisimProof

bisim2E  (C|||+ P Q) (inj1 x) = inj2 x
bisim2E  (C|||+ P Q) (inj2 y) = inj1 y
bisimELab (C|||+ P Q) (inj1 x) = refl
bisimELab (C|||+ P Q) (inj2 y) = refl
bisimENext (C|||+ P Q) (inj1 x) = C|||∞+ (PE P x) Q
bisimENext (C|||+ P Q) (inj2 y) = C|||∞+ P (PE Q y)
bisim2I  (C|||+ P Q) (inj1 x) = inj2 x
bisim2I  (C|||+ P Q) (inj2 y) = inj1 y
bisimINext (C|||+ P Q) (inj1 x) = C|||∞+ (PI P x) Q
bisimINext (C|||+ P Q) (inj2 y) = C|||∞+ P (PI Q y)
bisim2T  (C|||+ P Q) (x ,, x1) = (x1 ,, x)
bisim2TEq (C|||+ P Q) (x ,, x1) = refl
bisim2Er  (C|||+ P Q) (inj1 x) = inj2 x
bisim2Er  (C|||+ P Q) (inj2 y) = inj1 y
bisimELabr (C|||+ P Q) (inj1 x) = refl
bisimELabr (C|||+ P Q) (inj2 y) = refl
bisimENextr (C|||+ P Q) (inj1 x) = C|||∞+ P (PE Q x)
bisimENextr (C|||+ P Q) (inj2 y) = C|||∞+ (PE P y) Q
bisim2Ir (C|||+ P Q) (inj1 x) = inj2 x
bisim2Ir (C|||+ P Q) (inj2 y) = inj1 y
bisimINextr (C|||+ P Q) (inj1 x) = C|||∞+ P (PI Q x)
bisimINextr (C|||+ P Q) (inj2 y) = C|||∞+ (PI P y) Q
bisim2Tr  (C|||+ P Q) (x ,, x1) = x1 ,, x
bisim2TEqr (C|||+ P Q) (x ,, x1) = refl

--@END

C||| : {lu : LUniv} {c0 c1 : Choice} (P : Process ∞ {lu} c0) (Q : Process ∞ {lu} c1)

```

```

      → Bisims (P ||| Q) (fmap swap× (Q ||| P))
C||| (terminate x) (terminate x₁) = eqterminate
C||| (terminate x) (node x₁) = eqnode (lemBisimFmap+ (λ a → a „ x) swap× x₁)
C||| (node x) (terminate x₁) = eqnode (lemBisimFmap+ (λ „ → x₁) swap× x)
C||| (node P) (node Q) = eqnode (C|||+ P Q)

C|||∞ : {lu : LUniv}{c₀ c₁ : Choice} (P : Process∞ ∞ {lu} c₀) (Q : Process∞ ∞ {lu} c₁)
      → Bisims∞ (P |||∞ Q) (fmap∞ swap× (Q |||∞ P))
forceB (C|||∞ P Q) = C||| (forcep P) (forcep Q)

C|||+p : {lu : LUniv}{c₀ c₁ : Choice}
      → (P : Process+ ∞ {lu} c₀)
      → (Q : Process ∞ {lu} c₁)
      → Bisims+ (P |||+p Q) (fmap+ swap× (Q |||+p P))
C|||+p P (terminate x) = lemBisimFmap+ (λ „ → x) swap× P
C|||+p P (node x) = C|||+ P x

C|||p+ : {lu : LUniv}{c₀ c₁ : Choice}
      → (P : Process ∞ {lu} c₀)
      → (Q : Process+ ∞ {lu} c₁)
      → Bisims+ (P |||p+ Q) (fmap+ swap× (Q |||+p P))
C|||p+ (terminate x) Q = lemBisimFmap+ (λ a → a „ x) swap× Q
C|||p+ (node P) Q = C|||+ P Q

C|||+∞ : {lu : LUniv}{c₀ c₁ : Choice}
      → (P : Process+ ∞ {lu} c₀)
      → (Q : Process∞ ∞ {lu} c₁)
      → Bisims∞ (P |||+∞ Q) (fmap∞ swap× (Q |||∞+ P))
forceB (C|||+∞ P Q) = eqnode (C|||+p P (forcep Q))

C|||∞+ : {lu : LUniv}{c₀ c₁ : Choice}
      → (P : Process∞ ∞ {lu} c₀)
      → (Q : Process+ ∞ {lu} c₁)
      → Bisims∞ (P |||∞+ Q) (fmap∞ swap× (Q |||+∞ P))
forceB (C|||∞+ P Q) = eqnode (C|||p+ (forcep P) Q)

```

S



S

S

S

V

V

$$(\text{SW}||| P P') \ l \ m \ tr$$

--@END

```
--@BEGIN@SymIntBismiWbisimTraceEqForR
```

$$\begin{aligned} \text{WbisimTraceEqFor} \parallel r : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ \rightarrow (\text{fmap swap} \times (P' \parallel P)) \sqsubseteq (P \parallel P') \\ \text{WbisimTraceEqFor} \parallel r \ P \ P' \ l \ m \ tr = \text{bisimTraceEq} \ (\text{fmap swap} \times (P' \parallel P)) \\ (P \parallel P') \\ (\text{SW} \parallel r \ P \ P') \ l \ m \ tr \end{aligned}$$

--@END

```
--@BEGIN@SymIntBisimWbisimFdione
```

$$\begin{aligned} \text{WbisimFdi}_1 |||' : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ (P \text{ } P' : \text{Process} \infty \{lu\} c) \\ \rightarrow (P ||| P') \sqsubseteq_{\text{fdi}_1} (\text{fmap swap} \times (P' ||| P)) \\ \text{WbisimFdi}_1 |||' P \text{ } P' l x = \text{bisimImTrD} (P ||| P') \\ (\text{fmap swap} \times (P' ||| P)) \\ (\text{SW} ||| P \text{ } P') l x \end{aligned}$$

--@END

```
--@BEGIN@SymIntBisimWbisimFdioneR
```

$$\begin{aligned} & \text{WbisimFdi}_1 \parallel r : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & \quad (P \ P' : \text{Process} \ \infty \ \{lu\} \ c) \\ & \quad \rightarrow (\text{fmap swap} \times (P' \parallel P)) \sqsubseteq \text{fdi}_1 (P \parallel P') \\ & \text{WbisimFdi}_1 \parallel r \ P \ P' \ l \ x = \text{bisimImTrD} (\text{fmap swap} \times (P' \parallel P)) \\ & \quad (P \parallel P') \\ & \quad (\text{SW} \parallel r \ P \ P') \ l \ x \end{aligned}$$

--@END

```
--@BEGIN@SymIntBisimWbisimFditwo
```

```
WbisimFdi2|||' : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  → (P ||| P') ⊆fdi2ros (fmap swap × (P' ||| P))
WbisimFdi2|||' P P' l X x1 = bisimRefusalros (P ||| P')
  (fmap swap × (P' ||| P))
  (SW||| P P') l X x1
```

```
WbisimFdi3|||' : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  → (P ||| P') ⊆fdi3 (fmap swap × (P' ||| P))
WbisimFdi3|||' P P' = bisimImFDI3
  (P ||| P') (fmap swap × (P' ||| P)) (SW||| P P')
```

```
--@END
```

```
--@BEGIN@SymIntBisimWbisimFditwoR
```

```
WbisimFdi2|||r : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  → (fmap swap × (P' ||| P)) ⊆fdi2ros (P ||| P')
WbisimFdi2|||r P P' l X x1 = bisimRefusalros (fmap swap × (P' ||| P))
  (P ||| P')
  (SW|||r P P') l X x1
```

```
WbisimFdi3|||r : {lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  → (fmap swap × (P' ||| P)) ⊆fdi3 (P ||| P')
WbisimFdi3|||r P P' = bisimImFDI3
  (fmap swap × (P' ||| P)) (P ||| P') (SW|||r P P')
```

```
--@END
```

```
--@BEGIN@SymIntBisimWbisimFdiR
```

```

WbisimFdi|||r : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → (fmap swap× (P' ||| P)) ⊑fdi (P ||| P')
WbisimFdi|||r P P' = ((WbisimTraceEqFor|||r P P'
  „ WbisimFdi1|||r P P')
  „ WbisimFdi2|||r P P')
  „ WbisimFdi3|||r P P')

--@END

--@BEGIN@SymIntBisimWbisimFdi

WbisimFdi||| : {lu : LUniv}{c : Choice}(P P' : Process ∞ {lu} c)
  → (P ||| P') ⊑fdi (fmap swap× (P' ||| P))
WbisimFdi||| P P' = (((WbisimTraceEqFor|||' P P')
  „ (WbisimFdi1|||' P P'))
  „ (WbisimFdi2|||' P P'))
  „ (WbisimFdi3|||' P P'))

--@END

--@BEGIN@SymIntBisimWbisimFdiEq

WbisimFdiEq||| : {lu : LUniv}{lu : LUniv}{c : Choice}
  (P P' : Process ∞ {lu} c)
  → (P ||| P') ≡fdi (fmap swap× (P' ||| P))
WbisimFdiEq||| P P' = (WbisimFdi||| P P') „ WbisimFdi|||r P P'

--@END

--@BEGIN@CommutateInterleaveTrace

commute|||Trace+ : {lu : LUniv}{c0 c1 : Choice}
  (P : Process+ ∞ {lu} c0)
  (P' : Process+ ∞ {lu} c1)

```

$$\rightarrow (P \parallel\!\!\parallel P') \equiv (fmap\ swap \times (P' \parallel\!\!\parallel P))$$

--@END

--@BEGIN@CommuteInterleaveTraceProof

$$\begin{aligned} commute\parallel\!\!\parallel Trace\ P\ P' = & \text{bisimTraceEqs} \\ & (fmap\ swap \times (P' \parallel\!\!\parallel P)) \\ & (C\parallel\!\!\parallel P\ P') \end{aligned}$$

--@END

--@BEGIN@CommuteInterleaveSF

$$\begin{aligned} commute\parallel\!\!\parallel SF\ : \quad & \{lu : LUniv\} \{c_0\ c_1 : Choice\} \\ & (P : Process \infty \{lu\}\ c_0) \\ & (P' : Process \infty \{lu\}\ c_1) \\ \rightarrow (P \parallel\!\!\parallel P') = & sf\ (fmap\ swap \times (P' \parallel\!\!\parallel P)) \end{aligned}$$

--@END

--@BEGIN@CommuteInterleaveSFProof

$$\begin{aligned} commute\parallel\!\!\parallel SF\ P\ P' = & \text{bisimImplies} \\ & (fmap\ swap \times (P' \parallel\!\!\parallel P)) \\ & (C\parallel\!\!\parallel P\ P') \end{aligned}$$

--@END

--@BEGIN@CommuteInterleaveFDI

$$\begin{aligned} commute\parallel\!\!\parallel FDI\ : \quad & \{lu : LUniv\} \{c_0\ c_1 : Choice\} \\ & (P : Process \infty \{lu\}\ c_0) \\ & (P' : Process \infty \{lu\}\ c_1) \\ \rightarrow (P \parallel\!\!\parallel P') \equiv & fdi\ (fmap\ swap \times (P' \parallel\!\!\parallel P)) \end{aligned}$$

--@END

--@BEGIN@CommuteInterleaveFDIProof

```

commute|||FDI+ P P' = bisimslmFdiEquiv (P |||++ P')
                                (fmap+ swap× (P' |||++ P))
                                (C|||+ P P')

--@END

```

A.73 proofBisimSFFdiMonadicLaws.agda

```

--@PREFIX@proofBisimSFFdiMonadicLaws
-- was proofBisimImpliesFdimonadicLaw
{-# OPTIONS --allow-unsolved-metas #-}
module proofBisimSFFdiMonadicLaws where
-- was proofBisimSFFdiMonadicLaws.agda

open import process
open import choiceSetU
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Sum
open import auxData
open import RefWithoutSize
open import bisimilarity
open import bisimImpliesBisimw
open import bisimilarityProofs
open import bisimImpliesTraceEquiv
open import bisimImpliesFDI
open import fdiRefusal
open import bisimImpliesBisim
open import bisimSym
open import sequentialCompositionRev
open import labelUniv
open import bisimImpliesFDIPartTwo
open import bisimImpliesTraceEquiv
open import traceEquivalence renaming (_≡_ to _≡tr_; _≡+_ to _≡tr+_)
open import bisimwImpliesStableFailuresEquivalence

--@BEGIN@monadicLawones

monadicLaw1s : {lu : LUniv}{c0 c1 : Choice}

```

```

      (a : ChoiceSet c0)
      (P : ChoiceSet c0 → Process ∞ {lu} c1)
      → Bisims (terminate a >>= P) (P a)

--@END
--@BEGIN@monadicLawonesProof

monadicLaw1s a P = BismsRef (P a)

--@END

--@BEGIN@monadicLawone

monadicLaw1 : {lu : LUniv} {c0 c1 : Choice}
              (a : ChoiceSet c0)
              (P : ChoiceSet c0 → Process ∞ {lu} c1)
              → Bisimw (terminate a >>= P) (P a)

--@END

--@BEGIN@monadicLawoneProof
monadicLaw1 a P = BismwRef (P a)

--@END

--@BEGIN@monadicLawoneR

monadicLaw1R : {lu : LUniv} {c0 c1 : Choice}
              (a : ChoiceSet c0)
              (P : ChoiceSet c0 → Process ∞ {lu} c1)
              → Bisimw (P a) (terminate a >>= P)
monadicLaw1R a P = BismwRef (P a)

--@END

--@BEGIN@WbisimTraceEqForMonadone

```

```

WbisimTraceEqForMonad1 : {lu : LUniv} {c0 c1 : Choice}
    (a : ChoiceSet c0)
    (P : ChoiceSet c0 → Process ∞ {lu} c1)
    → (P a) ⊆ (terminate a >>= P)
WbisimTraceEqForMonad1 a P l m tr = bisimTraceEq (P a)
    (terminate a >>= P)
    (monadicLaw1 a P) l m tr

--@END

--@BEGIN@WbisimTraceEqForMonadoneR

WbisimTraceEqForMonad1r : {lu : LUniv}{c0 c1 : Choice}
    (a : ChoiceSet c0)
    (P : ChoiceSet c0 → Process ∞ {lu} c1)
    → (terminate a >>= P) ⊆ (P a)
WbisimTraceEqForMonad1r a P l m tr = bisimTraceEq
    (terminate a >>= P) (P a)
    (monadicLaw1R a P) l m tr

--@END

--@BEGIN@WbisimFdioneMonadone

WbisimFdi1Monad1 : {lu : LUniv} {c0 c1 : Choice}
    (a : ChoiceSet c0)
    (P : ChoiceSet c0 → Process ∞ {lu} c1)
    → (P a) ⊆ fdi1 (terminate a >>= P)
WbisimFdi1Monad1 a P l x = bisimImTrD (P a)
    (terminate a >>= P)
    (monadicLaw1 a P) l x

--@END

--@BEGIN@WbisimFdioneMonadoneR

WbisimFdi1Monad1r : {lu : LUniv}{c0 c1 : Choice}
    (a : ChoiceSet c0)
    (P : ChoiceSet c0 → Process ∞ {lu} c1)

```

```

      → (terminate a >>= P) ⊑fdi1 (P a)
WbisimFdi1Monad1r a P l x = bisimImTrD (terminate a >>= P)
      (P a)
      (monadicLaw1R a P) l x

```

```
--@END
```

```
--@BEGIN@WbisimFditwoMonadone
```

```

WbisimFdi2Monad1 : {lu : LUniv}{c0 c1 : Choice}
      (a : ChoiceSet c0)
      (P : ChoiceSet c0 → Process ∞ {lu} c1)
      → (P a) ⊑fdi2ros (terminate a >>= P)
WbisimFdi2Monad1 a P l X x = bisimRefusalros (P a)
      (terminate a >>= P)
      (monadicLaw1 a P) l X x

```

```

WbisimFdi3Monad1 : {lu : LUniv}{c0 c1 : Choice}
      (a : ChoiceSet c0)
      (P : ChoiceSet c0 → Process ∞ {lu} c1)
      → (P a) ⊑fdi3 (terminate a >>= P)
WbisimFdi3Monad1 a P = bisimImFDI3 (P a) (terminate a >>= P) (monadicLaw1 a P)

```

```
--@END
```

```
--@BEGIN@WbisimFditwoMonadoneR
```

```

WbisimFdi2Monad1r : {lu : LUniv}{c0 c1 : Choice}
      (a : ChoiceSet c0)
      (P : ChoiceSet c0 → Process ∞ {lu} c1)
      → (terminate a >>= P) ⊑fdi2ros (P a)
WbisimFdi2Monad1r a P l X x = bisimRefusalros
      (terminate a >>= P)
      (P a)(monadicLaw1R a P) l X x

```

```

WbisimFdi3Monad1r : {lu : LUniv}{c0 c1 : Choice}
  (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (terminate a ≥ P) ⊑fdi3 (P a)
WbisimFdi3Monad1r a P = bisimImFDI3 (terminate a ≥ P) (P a) (monadicLaw1R a P)

--@END

--@BEGIN@WbisimFdiMonadone

WbisimFdiMonad1 : {lu : LUniv}{c0 c1 : Choice}
  (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (P a) ⊑fdi (terminate a ≥ P)
WbisimFdiMonad1 a P = (((WbisimTraceEqForMonad1 a P)
  „ (WbisimFdi1Monad1 a P))
  „ (WbisimFdi2Monad1 a P))
  „ WbisimFdi3Monad1 a P

--@END

--@BEGIN@WbisimFdiMonadoneR

WbisimFdiMonad1r : {lu : LUniv}{c0 c1 : Choice}
  (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (terminate a ≥ P) ⊑fdi (P a)
WbisimFdiMonad1r a P = (((WbisimTraceEqForMonad1r a P)
  „ (WbisimFdi1Monad1r a P))
  „ (WbisimFdi2Monad1r a P))
  „ WbisimFdi3Monad1r a P

--@END

--@BEGIN@WbisimFdiEqMonadone

WbisimFdiEqMonad1 : {lu : LUniv}{c0 c1 : Choice}
  (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)

```

```

      → (terminate a >>= P) ≡ fdi (P a)
WbisimFdiEqMonad1 a P = (WbisimFdiMonad1 a P)
                        ,, (WbisimFdiMonad1r a P)

--@END

mutual
--@BEGIN@monadicLawthreeInf

monadicLaw1-3∞ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ Bisims∞ ((P >>=∞ Q) >>=∞ R)
  (P >>=∞ (λ x → Q x >>= R))

--@END

--@BEGIN@monadicLawthreeInfProof

forceB (monadicLaw1-3∞ P Q R) = monadicLaw1-3 (forcep P) Q R

monadicLaw1-3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ Bisims ((P >>= Q) >>= R)
  (P >>= (λ x → Q x >>= R))
monadicLaw1-3 (terminate x) Q R =
  BismsRef (((terminate x >>= Q) >>= R))
monadicLaw1-3 (node x) Q R = eqnode (monadicLaw1-3+ x Q R)

monadicLaw1-3+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)

```

```

(R : ChoiceSet c1 → Process ∞ {lu} c2)
→ Bisims+ ((P >>=+ Q) >>=+ R)
(P >>=+ (λ x → Q x >>= R))
bisim2E (monadicLaw1-3+ P Q R) e = e
bisimELab (monadicLaw1-3+ P Q R) e = refl
bisimENext (monadicLaw1-3+ P Q R) e =
  monadicLaw1-3∞ (PE P e) Q R
bisim2l (monadicLaw1-3+ P Q R) (inj1 (inj1 x)) = inj1 x
bisim2l (monadicLaw1-3+ P Q R) (inj1 (inj2 y)) = inj2 y
bisim2l (monadicLaw1-3+ P Q R) (inj2 ())
bisimlNext (monadicLaw1-3+ P Q R) (inj1 (inj1 x)) =
  monadicLaw1-3∞ (PI P x) Q R
bisimlNext (monadicLaw1-3+ P Q R) (inj1 (inj2 y)) =
  monadPT+ P Q R y
bisimlNext (monadicLaw1-3+ P Q R) (inj2 ())
bisim2T (monadicLaw1-3+ P Q R) ()
bisim2TEq (monadicLaw1-3+ P Q R) ()
bisim2Er (monadicLaw1-3+ P Q R) e = e
bisimELabr (monadicLaw1-3+ P Q R) e = refl
bisimENextr (monadicLaw1-3+ P Q R) e = monadicLaw∞ P Q R e
bisim2lr (monadicLaw1-3+ P Q R) (inj1 x) = inj1 (inj1 x)
bisim2lr (monadicLaw1-3+ P Q R) (inj2 y) = inj1 (inj2 y)
bisimlNextr (monadicLaw1-3+ P Q R) (inj1 x) = monadicLaw1-3∞ (PI P x) Q R
bisimlNextr (monadicLaw1-3+ P Q R) (inj2 y) =
  monadPT+ P Q R y
bisim2Tr (monadicLaw1-3+ P Q R) e = e
bisim2TEqr (monadicLaw1-3+ P Q R) ()

```

```
--@END
```

```
--@BEGIN@monadPTplus
```

```

monadPT+ : {lu : LUniv}{c0 c1 c2 : Choice}
(P : Process+ ∞ {lu} c0)
(Q : ChoiceSet c0 → Process ∞ {lu} c1)
→ (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ (y : ChoiceSet (T P))
→ Bisims∞ ((PI (P >>=+ Q) (inj2 y) >>=∞ R))
(PI (P >>=+ (λ x → Q x >>= R)) (inj2 y))
forceB (monadPT+ P Q R y) = BismsRef (Q (PT P y) >>= R)

```

```
--@END
```

```
--@BEGIN@monadicLawinf
```

```
monadicLaw $\infty$  : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+  $\infty$  {lu} c0)
  (Q : ChoiceSet c0  $\rightarrow$  Process  $\infty$  {lu} c1)
  (R : ChoiceSet c1  $\rightarrow$  Process  $\infty$  {lu} c2)
 $\rightarrow$  (x : ChoiceSet (E P))
 $\rightarrow$  Bisims $\infty$  (((PE P x  $\gg=\infty$  Q)  $\gg=\infty$  R))
  ((PE P x  $\gg=\infty$  ( $\lambda$  x  $\rightarrow$  Q x  $\gg=$  R))))
monadicLaw $\infty$  P Q R x = monadicLaw1-3 $\infty$  (PE P x) Q R
```

```
--@END
```

```
--@BEGIN@monadthreeSWplus
```

```
monad3SW+ : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+  $\infty$  {lu} c0)
  (Q : ChoiceSet c0  $\rightarrow$  Process  $\infty$  {lu} c1)
  (R : ChoiceSet c1  $\rightarrow$  Process  $\infty$  {lu} c2)
 $\rightarrow$  Bisimw+ ((P  $\gg=+$  Q)  $\gg=+$  R)
  (P  $\gg=+$  ( $\lambda$  x  $\rightarrow$  Q x  $\gg=$  R ))
```

```
--@END
```

```
--@BEGIN@monadthreeSWplusProof
```

```
monad3SW+ P Q R = bisimsToBismw+
  ((P  $\gg=+$  Q)  $\gg=+$  R)
  (P  $\gg=+$  ( $\lambda$  x  $\rightarrow$  Q x  $\gg=$  R))
  (monadicLaw1-3+ P Q R)
```

```
--@END
```

```
--@BEGIN@monadthreeSW
```

```
monad3SW : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process  $\infty$  {lu} c0)
```

```

      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
      (R : ChoiceSet c1 → Process ∞ {lu} c2)
    → Bisimw ((P      >>= Q) >>= R)
               (P      >>= (λ x → Q x >>= R ))
monad3SW P Q R = bisimsToBismw ((P >>= Q) >>= R)
                          (P >>= (λ x → Q x >>= R))
                          (monadicLaw1-3 P Q R)

```

```
--@END
```

```
--@BEGIN@monadthreeSWplusr
```

```

monad3SW+r : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → Bisimw+ (P      >>=+ (λ x → Q x >>= R ))
              ((P >>=+ Q) >>=+ R)
monad3SW+r P Q R = BismwSym+ ((P >>=+ Q) >>=+ R)
                      (P >>=+ (λ x → Q x >>= R))
                      (monad3SW+ P Q R)

```

```
--@END
```

```
--@BEGIN@monadthreeSWr
```

```

monad3SWr : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → Bisimw (P      >>= (λ x → Q x >>= R ))
              ((P      >>= Q) >>= R)
monad3SWr P Q R = BismwSym ((P >>= Q) >>= R)
                      (P >>= (λ x → Q x >>= R))
                      (monad3SW P Q R)

```

```
--@END
```

```
--@BEGIN@wbisimTraceEqForMonadthree
```

```

WbisimTraceEqForMonad3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>= Q) >>= R) ⊆ (P >>= (λ x → Q x >>= R))
WbisimTraceEqForMonad3 P Q R l m tr = bisimTraceEq ((P >>= Q) >>= R)
  ((P >>= (λ x → Q x >>= R)))
  (monad3SW P Q R) l m tr

```

```
--@END
```

```
--@BEGIN@WbisimTraceEqForMonadthreeer
```

```

WbisimTraceEqForMonad3r : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P >>= (λ x → Q x >>= R)) ⊆ ((P >>= Q) >>= R)
WbisimTraceEqForMonad3r P Q R l m tr = bisimTraceEq
  ((P >>= (λ x → Q x >>= R)))
  ((P >>= Q) >>= R) (monad3SWr P Q R) l m tr

```

```
--@END
```

```
--@BEGIN@WbisimFdioneMonadthree
```

```

WbisimFdi1Monad3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>= Q) >>= R) ⊆ fdi1
  (P >>= (λ x → Q x >>= R))
WbisimFdi1Monad3 P Q R l x = bisimImTrD ((P >>= Q) >>= R)
  (P >>= (λ x1 → Q x1 >>= R))
  (monad3SW P Q R) l x

```

--@END

--@BEGIN@WbisimFdioneMonadthree

```

WbisimFdi1Monad3r : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ (P >>= (λ x → Q x >>= R)) ⊑fdi1
  ((P >>= Q) >>= R)
WbisimFdi1Monad3r P Q R l x = bisimImTrD
  (P >>= (λ x1 → Q x1 >>= R))
  ((P >>= Q) >>= R) (monad3SWr P Q R) l x

```

--@END

--@BEGIN@WbisimFditwoMonadthree

```

WbisimFdi2Monad3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ ((P >>= Q) >>= R) ⊑fdi2ros
  (P >>= (λ x → Q x >>= R))
WbisimFdi2Monad3 P Q R l X x = bisimRefusalros
  ((P >>= Q) >>= R)
  (P >>= (λ x1 → Q x1 >>= R))
  (monad3SW P Q R) l X x

```

```

WbisimFdi3Monad3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)

```

```

→ ((P >>= Q) >>= R) ⊑fdi3
  (P >>= (λ x → Q x >>= R))
WbisimFdi3Monad3 P Q R = bisimImFDI3
  ((P >>= Q) >>= R)
  (P >>= (λ x → Q x >>= R))
  (monad3SW P Q R)

```

```
--@END
```

```
--@BEGIN@WbisimFditwoMonadthree
```

```

WbisimFdi2Monad3r : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ (P >>= (λ x → Q x >>= R)) ⊑fdi2ros
  ((P >>= Q) >>= R)
WbisimFdi2Monad3r P Q R l X x = bisimRefusalros
  (P >>= (λ x1 → Q x1 >>= R))
  ((P >>= Q) >>= R)
  ((monad3SWr P Q R)) l X x

```

```

WbisimFdi3Monad3r : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
→ (P >>= (λ x → Q x >>= R)) ⊑fdi3
  ((P >>= Q) >>= R)
WbisimFdi3Monad3r P Q R = bisimImFDI3
  (P >>= (λ x → Q x >>= R))
  ((P >>= Q) >>= R)
  (monad3SWr P Q R)

```

```
--@END
```

```
--@BEGIN@WbisimFdiMonadthree
```

```
WbisimFdiMonad3 : {lu : LUniv}{c0 c1 c2 : Choice}
```

```

(P : Process ∞ {lu} c₀)
(Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
(R : ChoiceSet c₁ → Process ∞ {lu} c₂)
→ ((P >>= Q) >>= R) ⊑fdi
(P >>= (λ x → Q x >>= R))

```

```

WbisimFdiMonad₃ P Q R = (((WbisimTraceEqForMonad₃ P Q R)
  „ (WbisimFdi₁Monad₃ P Q R))
  „ (WbisimFdi₂Monad₃ P Q R))
  „ WbisimFdi₃Monad₃ P Q R

```

```
--@END
```

```
--@BEGIN@WbisimFdiMonadthree
```

```

WbisimFdiMonad₃r : {lu : LUniv}{c₀ c₁ c₂ : Choice}
(P : Process ∞ {lu} c₀)
(Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
(R : ChoiceSet c₁ → Process ∞ {lu} c₂)
→ (P >>= (λ x → Q x >>= R)) ⊑fdi
((P >>= Q) >>= R)

```

```

WbisimFdiMonad₃r P Q R = (((WbisimTraceEqForMonad₃r P Q R)
  „ (WbisimFdi₁Monad₃r P Q R))
  „ (WbisimFdi₂Monad₃r P Q R))
  „ WbisimFdi₃Monad₃r P Q R

```

```
--@END
```

```
--@BEGIN@WbisimFdiEqMonadthree
```

```

WbisimFdiEqMonad₃ : {lu : LUniv}{c₀ c₁ c₂ : Choice}
(P : Process ∞ {lu} c₀)
(Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
(R : ChoiceSet c₁ → Process ∞ {lu} c₂)
→ ((P >>= Q) >>= R) ≡fdi (P >>= (λ x → Q x >>= R))

```

```

WbisimFdiEqMonad₃ P Q R = (WbisimFdiMonad₃ P Q R)
  „ (WbisimFdiMonad₃r P Q R)

```

```
--@END
```

```
--@BEGIN@monadicLawOneTrace
```

```
monadicLaw1Trace : {lu : LUniv}{c0 c1 : Choice}
                  (a : ChoiceSet c0)
                  (P : ChoiceSet c0 → Process ∞ {lu} c1)
                  → (terminate a >>= P) ≡tr (P a)
```

```
--@END
```

```
--@BEGIN@monadicLawOneTraceProof
```

```
monadicLaw1Trace a P =      bisimTraceEq=
                              (terminate a >>= P) (P a) (monadicLaw1 a P)
```

```
--@END
```

```
--@BEGIN@monadicLawOneSF
```

```
monadicLaw1SF+ : {lu : LUniv}{c0 c1 : Choice}
                (a : ChoiceSet c0)
                (P : ChoiceSet c0 → Process ∞ {lu} c1)
                → (terminate a >>= P) =sf (P a)
```

```
--@END
```

```
--@BEGIN@monadicLawOneSFProof
```

```
monadicLaw1SF+ a P =      bisimwImplies=sf
                              (terminate a >>= P) (P a) (monadicLaw1 a P)
```

```
--@END
```

```
--@BEGIN@monadicLawOneFDI
```

```
monadicLaw1FDI+ : {lu : LUniv}{c0 c1 : Choice}
                  (a : ChoiceSet c0)
```

$$(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) \\ \rightarrow (\text{terminate } a \gg= P) \equiv \text{fdi } (P a)$$

--@END

--@BEGIN@monadicLawOneFDIProof

$$\text{monadicLaw}_1 \text{FDI} + a P = \quad \text{bisimFDIImpEq} \\ (\text{terminate } a \gg= P) (P a) (\text{monadicLaw}_1 a P)$$

--@END

--@BEGIN@monadicLawThreeTrace

$$\text{monadicLaw}_3 \text{Trace} + : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\ (P : \text{Process} + \infty \{lu\} c_0) \\ (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) \\ (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) \\ \rightarrow ((P \gg= + Q) \gg= + R) \equiv \text{tr} + \\ (P \gg= + (\lambda x \rightarrow Q x \gg= R))$$

--@END

--@BEGIN@monadicLawThreeTraceProof

$$\text{monadicLaw}_3 \text{Trace} + P Q R = \quad \text{bisimTraceEqs} + = \\ ((P \gg= + Q) \gg= + R) \\ (P \gg= + (\lambda x \rightarrow Q x \gg= R)) \\ (\text{monadicLaw}_{1-3} + P Q R)$$

--@END

--@BEGIN@monadicLawThreeSF

$$\text{monadicLaw}_3 \text{SF} + : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\ (P : \text{Process} + \infty \{lu\} c_0) \\ (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) \\ (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2)$$

$$\rightarrow ((P \gg=+ Q) \gg=+ R) =_{\text{sf}}+ \\ (P \gg=+ (\lambda x \rightarrow Q x \gg= R))$$

--@END

--@BEGIN@monadicLawThreeSFProof

$$\text{monadicLaw}_3\text{SF}+ P Q R = \text{bisimslmImplies} =_{\text{sf}}+ \\ ((P \gg=+ Q) \gg=+ R) \\ (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\ (\text{monadicLaw}_{1-3}+ P Q R)$$

--@END

--@BEGIN@monadicLawThreeFDI

$$\text{monadicLaw}_3\text{FDI}+ : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} \\ (P : \text{Process}+ \infty \{lu\} c_0) \\ (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \{lu\} c_1) \\ (R : \text{ChoiceSet } c_1 \rightarrow \text{Process} \infty \{lu\} c_2) \\ \rightarrow ((P \gg=+ Q) \gg=+ R) \equiv_{\text{fdi}}+ \\ (P \gg=+ (\lambda x \rightarrow Q x \gg= R))$$

--@END

--@BEGIN@monadicLawThreeFDIProof

$$\text{monadicLaw}_3\text{FDI}+ P Q R = \text{bisimslmFdiEquiv} \\ ((P \gg=+ Q) \gg=+ R) \\ (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) \\ (\text{monadicLaw}_{1-3}+ P Q R)$$

--@END

A.74 proofBisimSFFdiMonadicLaws2.agda

--@PREFIX@proofBisimSFFdiMonadicLawsTwo

```

module proofBisimSFFdiMonadicLaws2 where

open import process
open import choiceSetU
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Sum
open import auxData
open import RefWithoutSize
open import bisimilarity
open import bisimSImpliesBisimw
open import bisimilarityProofs
open import bisimSImpliesTraceEquiv
open import bisimSImpliesFDI
open import fdiRefusal
open import bisimSImpliesBisim
open import bisimSym
open import sequentialCompositionRev
open import labelUniv
open import bisimSImpliesFDIPartTwo
open import bisimSImpliesTraceEquiv
open import traceEquivalence renaming (_≡_ to _≡tr_; _≡+_ to _≡tr+_)
open import bisimwImpliesStableFailuresEquivalence
open import dataAuxFunction
open import Data.Unit
open import Data.Empty

mutual
--@BEGIN@NoTicks

record NoTicks∞ {i : Size}
  {lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c) : Set where

  coinductive
  field
    forceNT : {j : Size < i} → NoTicks {j} (forcep P {∞})

NoTicks : {i : Size}
  {lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)

```

```

    → Set
NoTicks (terminate x) = T
NoTicks (node P) = NoTicks+ P

record NoTicks+ {i : Size}
  {lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c) : Set where
  coinductive
  field
    noT : ¬ (ChoiceSet (T P))
    nextE : (ext1 : ChoiceSet (E P)) → NoTicks∞ (PE P ext1)
    nextI : (int1 : ChoiceSet (I P)) → NoTicks∞ (PI P int1)

--@END
open NoTicks∞ public
open NoTicks+ public

mutual

--@BEGIN@proofMonadicLawTwo

monadicLaw₂∞ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process∞ ∞ {lu} c)
  (noticks : NoTicks∞ {i} P)
  → Bisims∞ {i} (P >>=∞ terminate) P
forceB (monadicLaw₂∞ {i} {lu} {c} P noticks) {j} =
  monadicLaw₂ {j} (forceP P {∞}) (forceNT noticks {j})

monadicLaw₂ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  (noticks : NoTicks {i} P)
  → Bisims {i} (P >>= terminate) P
monadicLaw₂ {i} (terminate x) noticks = eqterminate
monadicLaw₂ {i} (node P) noticks = eqnode (monadicLaw₂+ {i} P noticks)

monadicLaw₂+ : {i : Size}{lu : LUniv}{c : Choice}
  (P : Process+ ∞ {lu} c)
  (noticks : NoTicks+ {i} P)
  → Bisims+ {i} (P >>=+ terminate) P
bisim2E (monadicLaw₂+ P noticks) e = e

```



```

bisimELab (monadicLaw2+ P noticks) e = refl
bisimENext (monadicLaw2+ P noticks) e = monadicLaw2∞ (PE P e) (nextE noticks e)
bisim2l (monadicLaw2+ P noticks) (inj1 int1) = int1
bisim2l (monadicLaw2+ P noticks) (inj2 t) = ⊥-elim (noT noticks t)
bisimlNext (monadicLaw2+ P noticks) (inj1 int1) = monadicLaw2∞ (PI P int1) (nextl noticks int1)
bisimlNext (monadicLaw2+ P noticks) (inj2 t) = ⊥-elim (noT noticks t)
bisim2T (monadicLaw2+ P noticks) ()
bisim2TEq (monadicLaw2+ P noticks) ()
bisim2Er (monadicLaw2+ P noticks) e = e
bisimELabr (monadicLaw2+ P noticks) e = refl
bisimENextr (monadicLaw2+ P noticks) e = monadicLaw2∞ (PE P e) (nextE noticks e)
bisim2lr (monadicLaw2+ P noticks) int1 = inj1 int1
bisimlNextr (monadicLaw2+ {i} P noticks) int1
    = monadicLaw2∞ {i} (PI P int1) (nextl noticks int1)
bisim2Tr (monadicLaw2+ P noticks) t = ⊥-elim (noT noticks t)
bisim2TEqr (monadicLaw2+ P noticks) t = ⊥-elim (noT noticks t)

--@END

```

A.75 proofCommutativeExternalChoice.agda

```

--@PREFIX@proofCommutativeExternalChoice
--old: proofBisimImpliesComExternalChoice
--old: proofBisimImpliesSymExternalChoice

```

```

module proofCommutativeExternalChoice where

```

```

open import process
open import choiceSetU
open import Size
open import Relation.Binary.PropositionalEquality
open import Data.Sum
open import renamingResult
open import lemFmap
open import auxData
open import RefWithoutSize
open import bisimilarity
open import bisimImpliesBisimw
open import bisimilarityProofs

```



```

open import bisimImpliesTraceEquiv
open import bisimLemFmap
open import bisimImpliesFDI
open import fdiRefusal
open import bisimImpliesBisim
open import externalChoice
open import addTick
open import labelUniv
open import bisimImpliesFDIPartTwo
open import bisimImpliesTraceEquiv
open import traceEquivalence
open import bisimwImpliesStableFailuresEquivalence

mutual

--@BEGIN@ExtBisim

C□++ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
      → Bisims+ (P □++ Q) (fmap+ swap⊕ (Q □++ P))

--@END

--@BEGIN@ExtBisimProof

bisim2E   (C□++ P Q) (inj1 x) = inj2 x
bisim2E   (C□++ P Q) (inj2 y) = inj1 y
bisimELab (C□++ P Q) (inj1 x) = refl
bisimELab (C□++ P Q) (inj2 y) = refl
bisimENext (C□++ P Q) (inj1 x) =
  lemBisimFmap∞ inj2 swap⊕ (PE P x)
bisimENext (C□++ P Q) (inj2 y) =
  lemBisimFmap∞ inj1 swap⊕ (PE Q y)
bisim2I   (C□++ P Q) (inj1 x) = inj2 x
bisim2I   (C□++ P Q) (inj2 y) = inj1 y
bisimINext (C□++ {lu} P Q) (inj1 x) =
  C□∞++ {lu = lu} (PI P x) Q
bisimINext (C□++ {lu} P Q) (inj2 y) =
  C□+∞+ {lu = lu} P (PI Q y)
bisim2T   (C□++ P Q) (inj1 x) = inj2 x
bisim2T   (C□++ P Q) (inj2 y) = inj1 y

```

```

bisim2TEq (C□++ P Q) (inj1 x) = refl
bisim2TEq (C□++ P Q) (inj2 y) = refl
bisim2Er  (C□++ P Q) (inj1 x) = inj2 x
bisim2Er  (C□++ P Q) (inj2 y) = inj1 y
bisimELabr (C□++ P Q) (inj1 x) = refl
bisimELabr (C□++ P Q) (inj2 y) = refl
bisimENextr (C□++ P Q) (inj1 x) =
    lemBisimFmap∞ inj1 swap⊕ (PE Q x)
bisimENextr (C□++ P Q) (inj2 y) =
    lemBisimFmap∞ inj2 swap⊕ (PE P y)
bisim2lr   (C□++ P Q) (inj1 x) = inj2 x
bisim2lr   (C□++ P Q) (inj2 y) = inj1 y
bisimlNextr (C□++ P Q) (inj1 x) =
    C□+∞+ P (PI Q x)
bisimlNextr (C□++ P Q) (inj2 y) =
    C□∞++ (PI P y) Q
bisim2Tr   (C□++ P Q) (inj1 x) = inj2 x
bisim2Tr   (C□++ P Q) (inj2 y) = inj1 y
bisim2TEqr (C□++ P Q) (inj1 x) = refl
bisim2TEqr (C□++ P Q) (inj2 y) = refl

--@END

C□+∞+ : {lu : LUniv}{c0 c1 : Choice}
    → (P : Process+ ∞ {lu} c0)
    → (Q : Process∞ ∞ {lu} c1)
    → Bisims∞ (P □+∞+ Q) (fmap∞ swap⊕ (Q □∞++ P))
forceB (C□+∞+ P Q) = eqnode (C□+p+ P (forcep Q))

C□+p+ : {lu : LUniv}{c0 c1 : Choice}
    → (P : Process+ ∞ {lu} c0)
    → (Q : Process ∞ {lu} c1)
    → Bisims+ (P □+p+ Q) (fmap+ swap⊕ (Q □p++ P))
C□+p+ P (terminate x) = addTimeFmapBisimLemma+ inj2 swap⊕ P (inj1 x)
C□+p+ P (node Q) = C□++ P Q

C□∞++ : {lu : LUniv}{c0 c1 : Choice}
    → (P : Process∞ ∞ {lu} c0)

```

```

    → (Q : Process+ ∞ {lu} c₁)
    → Bisims∞ (P □∞++ Q) (fmap∞ swap⊕ (Q □+∞+ P))
forceB (C□∞++ P Q) = eqnode (C□p++ (forceP P) Q)

C□p++ : {lu : LUniv}{c₀ c₁ : Choice}
    → (P : Process ∞ {lu} c₀)
    → (Q : Process+ ∞ {lu} c₁)
    → Bisims+ (P □p++ Q) (fmap+ swap⊕ (Q □+p+ P))
C□p++ (terminate x) Q = addTimeFmapBisimLemma+ inj₁ swap⊕ Q (inj₂ x)
C□p++ (node P) Q = C□++ P Q

--@BEGIN@ComExtBisim

SW□+ : {lu : LUniv}{c₀ c₁ : Choice}
    (P : Process+ ∞ {lu} c₀)
    (Q : Process+ ∞ {lu} c₁)
    → Bisimw+ (P □++ Q) (fmap+ swap⊕ (Q □++ P))

--@END

--@BEGIN@ComExtBisimProof

SW□+ P Q = bisimsToBismw+ (P □++ Q)
    (fmap+ swap⊕ (Q □++ P)) (C□++ P Q)

--@END

--@BEGIN@ComExtBisimR

SW□+r : {lu : LUniv}{c₀ c₁ : Choice}
    (P : Process+ ∞ {lu} c₀)
    (Q : Process+ ∞ {lu} c₁)
    → Bisimw+ (fmap+ swap⊕ (Q □++ P)) (P □++ Q)
SW□+r P Q = BismwSym+ (P □++ Q)
    (fmap+ swap⊕ (Q □++ P)) (SW□+ P Q)

--@END

```

```
--@BEGIN@WbisimTraceEqForComExtBisim
```

```
WbisimTraceEqFor□+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (P □++ P') ⊑+ (fmap+ swap⊕ (P' □++ P))
WbisimTraceEqFor□+ P P' l m tr = bisimTraceEq+ (P □++ P')
  (fmap+ swap⊕ (P' □++ P))
  (SW□+ P P') l m tr
```

```
--@END
```

```
--@BEGIN@WbisimTraceEqForComExtBisimR
```

```
WbisimTraceEqFor□+r : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (fmap+ swap⊕ (P' □++ P)) ⊑+ (P □++ P')
WbisimTraceEqFor□+r P P' l m tr = bisimTraceEq+
  (fmap+ swap⊕ (P' □++ P))
  (P □++ P')
  (SW□+r P P') l m tr
```

```
--@END
```

```
--@BEGIN@WbisimFdioneComExtBisim
```

```
WbisimFdi₁□+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (P □++ P') ⊑fdi₁+ (fmap+ swap⊕ (P' □++ P))
WbisimFdi₁□+ P P' l x = bisimImTrD+
  (P □++ P')
  (fmap+ swap⊕ (P' □++ P))
  (SW□+ P P') l x
```

```
--@END
```

```
--@BEGIN@WbisimFdioneComExtBisimR
```

```
WbisimFdi₁□+r : {lu : LUniv}{c : Choice}
```

```

      (P P' : Process+ ∞ {lu} c)
      → (fmap+ swap⊕ (P' □++ P)) ⊑fdi1+ (P □++ P')
WbisimFdi1□+r P P' l x = bisimImTrD+
      (fmap+ swap⊕ (P' □++ P))
      (P □++ P')
      (SW□+r P P') l x

```

```
--@END
```

```
--@BEGIN@WbisimFditwoComExtBisim
```

```

WbisimFdi2□+ : {lu : LUniv}{c : Choice}
      (P P' : Process+ ∞ {lu} c)
      → (P □++ P') ⊑fdi2ros+ (fmap+ swap⊕ (P' □++ P))
WbisimFdi2□+ P P' l X x1 = bisimRefusalros+
      (P □++ P')
      (fmap+ swap⊕ (P' □++ P))
      (SW□+ P P') l X x1

```

```

WbisimFdi3□+ : {lu : LUniv}{c : Choice}
      (P P' : Process+ ∞ {lu} c)
      → (P □++ P') ⊑fdi3+ (fmap+ swap⊕ (P' □++ P))
WbisimFdi3□+ P P' = bisimImFDI3+
      (P □++ P')
      (fmap+ swap⊕ (P' □++ P))
      (SW□+ P P')

```

```
--@END
```

```
--@BEGIN@WbisimFditwoComExtBisimR
```

```

WbisimFdi2□+r : {lu : LUniv}{c : Choice}
      (P P' : Process+ ∞ {lu} c)
      → (fmap+ swap⊕ (P' □++ P)) ⊑fdi2ros+ (P □++ P')
WbisimFdi2□+r P P' l X x1 = bisimRefusalros+
      (fmap+ swap⊕ (P' □++ P))
      (P □++ P')
      (SW□+r P P') l X x1

```

```

WbisimFdi3□+r : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (fmap+ swap⊕ (P' □++ P)) ⊑fdi3+ (P □++ P')
WbisimFdi3□+r P P'      = bisimImFDI3+
  (fmap+ swap⊕ (P' □++ P))
  (P □++ P')
  (SW□+r P P')

```

```
--@END
```

```
--@BEGIN@WbisimFdiComExtBisimR
```

```

WbisimFdi□+r : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (fmap+ swap⊕ (P' □++ P)) ⊑fdi+ (P □++ P')
WbisimFdi□+r P P' = ((WbisimTraceEqFor□+r P P'
  „ WbisimFdi1□+r P P')
  „ WbisimFdi2□+r P P')
  „ WbisimFdi3□+r P P')

```

```
--@END
```

```
--@BEGIN@WbisimFdiComExtBisim
```

```

WbisimFdi□+ : {lu : LUniv}{c : Choice}
  (P P' : Process+ ∞ {lu} c)
  → (P □++ P') ⊑fdi+ (fmap+ swap⊕ (P' □++ P))
WbisimFdi□+ P P' = (((WbisimTraceEqFor□+ P P')
  „ (WbisimFdi1□+ P P'))
  „ (WbisimFdi2□+ P P'))
  „ WbisimFdi3□+ P P')

```

```
--@END
```

```
--@BEGIN@WbisimFdiComExtBisimTwoPart
```

```

WbisimFdiEq□+ : {lu : LUniv}{c : Choice}
                (P P' : Process+ ∞ {lu} c)
                → (P □++ P') ≡fdi+ (fmap+ swap⊕ (P' □++ P))
WbisimFdiEq□+ P P' = (WbisimFdi□+ P P') „ WbisimFdi□+r P P'

--@END

--@BEGIN@CommuteExtChWeakBisim

commuteExtChWeakBisim+ : {lu : LUniv}{c0 c1 : Choice}
                        (P : Process+ ∞ {lu} c0)
                        (P' : Process+ ∞ {lu} c1)
                        → Bisimw+ (P □++ P') (fmap+ swap⊕ (P' □++ P))
commuteExtChWeakBisim+ P P' = bisimsToBismw+ (P □++ P')
                        (fmap+ swap⊕ (P' □++ P))
                        (C□++ P P')

--@END

--@BEGIN@CommuteExtChTrace

commuteExtChTrace+ : {lu : LUniv}{c0 c1 : Choice}
                    (P : Process+ ∞ {lu} c0)
                    (P' : Process+ ∞ {lu} c1)
                    → (P □++ P') ≡+ (fmap+ swap⊕ (P' □++ P))

--@END

--@BEGIN@CommuteExtChTraceProof

commuteExtChTrace+ P P' = bisimTraceEqs+= (P □++ P')
                        (fmap+ swap⊕ (P' □++ P))
                        (C□++ P P')

--@END

--@BEGIN@CommuteExtChSF

commuteExtChSF+ : {lu : LUniv}{c0 c1 : Choice}

```

```

(P : Process+ ∞ {lu} c₀)
(P' : Process+ ∞ {lu} c₁ )
→ (P □++ P') =sf+ (fmap+ swap⊕ (P' □++ P))

--@END

--@BEGIN@CommutExtChSFProof

commuteExtChSF+ P P' =   bisimslmplies=sf+ (P □++ P')
                        (fmap+ swap⊕ (P' □++ P))
                        (C□++ P P')

--@END

--@BEGIN@CommutExtChFDI

commuteExtChFDI+ : {lu : LUniv}{c₀ c₁ : Choice}
                  (P : Process+ ∞ {lu} c₀)
                  (P' : Process+ ∞ {lu} c₁ )
                  → (P □++ P') ≡fdi+ (fmap+ swap⊕ (P' □++ P))

--@END

--@BEGIN@CommutExtChFDIProof

commuteExtChFDI+ P P' = bisimslmFdiEquiv (P □++ P')
                        (fmap+ swap⊕ (P' □++ P))
                        (C□++ P P')

--@END

```

A.76 proofEqforParSym.agda

```
--@PREFIX@mainproofEqforParSym
```

```
module proofEqforParSym where
```

```
open import process
```

```

open import Size
open import choiceSetU
open import renamingResult
open import lemFmap
open import Data.Product
open import labelUniv
open import parallelSimple
open import Data.Bool
open import traceEquivalence
open import proofSymParPartone
open import proofSymParR

--@BEGIN@EqforParSymDef

≡S[|]|+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (A B : Label lu → Bool)
      (Q : Process+ ∞ {lu} c1)
      → (P [ A ]|[ B ] Q) ≡+ (fmap+ swap× ((Q [ B ]|[ A ] P)))
≡S[|]|+ P A B Q = (S[|]|+ P A B Q) , (S[|]|+R P A B Q)

--@END

```

A.77 proofEqforParSymTheoOnly.agda

```

--@PREFIX@mainproofEqforParSymTheoOnly

module proofEqforParSymTheoOnly where

open import process
open import Size
open import choiceSetU
open import renamingResult
open import lemFmap
open import Data.Product
open import labelUniv
open import parallelSimple
open import Data.Bool

```

```

open import traceEquivalence
open import proofSymParPartone
open import proofSymParR

--@BEGIN@EqforParSymDef

≡S[|]|+ : {lu : LUniv}{c0 c1 : Choice}
          (P : Process+ ∞ {lu} c0)
          (A B : Label lu → Bool)
          (Q : Process+ ∞ {lu} c1)
          → (P [ A ]||+[ B ] Q) ≡+ (fmap+ swap× ((Q [ B ]||+[ A ] P)))

--@END

≡S[|]|+ P A B Q = (S[|]|+ P A B Q) , (S[|]|+R P A B Q)

```

A.78 ProofLawsSeq.agda

```

--@PREFIX@mainProofLawsSeq

module ProofLawsSeq where

open import process
open import Size
open import Level
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.Product
open import Data.Fin
open import Data.List
open import Data.Sum
open import labelUniv
open import dataAuxFunction
open import externalChoice
open import sequentialCompositionRev
open import renamingResult
open import TraceWithoutSize

```

```

open import RefWithoutSize
open import primitiveProcess
open import traceEquivalence
open import Data.Product

```

```

lemTrTerminateBind : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c)(x : ChoiceSet (T P))
  → Tr∞ [] (just (PT P x)) (PI (P >>=+ terminate) (inj₂ x))
lemTrTerminateBind c P x = ter (PT P x)

```

```

lemTrTerminateBind' : {lu : LUniv}(c₀ c₁ c₂ : Choice)
  (P : Process+ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ c₁)
  (R : ChoiceSet c₁ → Process ∞ c₂)
  (x : Fin 0)
  → Tr∞ [] (just (PT (λ x₁ → _>>=+ P (λ x₁ → _>>= (Q x₁) R)) x)) (PI (λ x₁ → _>>=+ P (λ x₁ → _>>= (Q x₁) R)) x))
lemTrTerminateBind' c P Q R x q ()

```

```
--@BEGIN@stopSeq
```

```

stopSeq : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
  (P : ChoiceSet c₀ → Process ∞ {lu} c₀)
  → (STOP c₀ >>= P) ⊆ STOP c₀
stopSeq a P .[] .nothing (tnode empty) = tnode empty
stopSeq a P .(efq _ :: l) m (tnode (extc l .m () x₁))
stopSeq a P l m (tnode (intc l .m () x₁))
stopSeq a P .[] .(just (efq _)) (tnode (terc ()))

```

```
--@END
```

```

stopSeqr : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
  (P : ChoiceSet c₀ → Process ∞ {lu} c₀)
  → STOP c₀ ⊆ (STOP c₀ >>= P)
stopSeqr a P .[] .nothing (tnode empty) = tnode empty
stopSeqr a P .(efq _ :: l) m (tnode (extc l .m () x₁))

```

```

stopSeqr a P l m (tnode (intc .l .m (inj1 ()) x1))
stopSeqr a P l m (tnode (intc .l .m (inj2 ()) x1))
stopSeqr a P .[] .(just (PT (process+ (fin 0) efq efq
                             (fin 0) efq (fin 0) efq "STOP"
                             >>=+ P) -)) (tnode (terc ()))

```

```

--@BEGIN@stopSeqEq

```

```

≡stopSeq : {lu : LUniv}{c0 : Choice} (a : ChoiceSet c0)
          (P : ChoiceSet c0 → Process ∞ {lu} c0)
          → STOP c0 {lu} ≡ (STOP c0 {lu} >>= P)
≡stopSeq a P = (stopSeqr a P) , (stopSeq a P)

```

```

--@END

```

```

--@BEGIN@unitSeqL

```

```

unitSeqL : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
          (P : ChoiceSet c0 → Process ∞ {lu} c1)
          → (terminate a >>= P) ⊆ P a
unitSeqL a P l m q = q

```

```

--@END

```

```

unitSeqLr : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)(P : ChoiceSet c0 → Process ∞ {lu} c1)
          → P a ⊆ (terminate a >>= P)
unitSeqLr {lu} {c0} {c1} a P l m q = q

```

```

--@BEGIN@unitSeqLEq

```

```

≡unitSeq : {lu : LUniv}{c0 : Choice} (a : ChoiceSet c0)

```

```

(P : ChoiceSet c0 → Process ∞ {lu} c0)
  → P a ≡ (terminate a ≫= P)
≡ unitSeq a P = (unitSeqL a P) , unitSeqLr a P

```

```
--@END
```

```

lemTrTerminateBind" : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c) (x : Fin 0)
  → Tr+ {lu} [] (just (PT (P ≫=+ terminate) x)) P
lemTrTerminateBind" c P ()

```

```

lemtr+trterminate : {lu : LUniv}{c0 : Choice} → (m : Maybe (ChoiceSet c0)) → (P : Process+ ∞ {lu} c0)
  (y : ChoiceSet (T P)) → (traux : Tr {lu} {c0} l m (terminate (PT P y))) → Tr {lu} {c0} l m (just (PT (P ≫=+ terminate) x))
lemtr+trterminate c0 .(just (PT P y)) P .[] y (ter .(PT P y)) = terc y
lemtr+trterminate c0 .nothing P .[] y (empty .(PT P y)) = empty

```

mutual

```

unitSeqR : {lu : LUniv}{c0 : Choice} (P : Process ∞ {lu} c0)
  → (P ≫= terminate) ⊆ P
unitSeqR (terminate x) l m q = q
unitSeqR (node x) l m (tnode q) = tnode (unitSeqR+ x l m q)

```

```
--@BEGIN@unitSeqR
```

```

unitSeqR+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)
  → (P ≫=+ terminate) ⊆+ P
unitSeqR+ P .[] .nothing empty
  = empty
unitSeqR+ P .(Lab P x :: l) m (extc l .m x x1)
  = extc l m x (unitSeqR∞ (PE P x) l m x1)
unitSeqR+ P l m (intc .l .m x x1)
  = intc l m (inj1 x) (unitSeqR∞ (PI P x) l m x1)
unitSeqR+ {lu} {c0} P .[] .(just (PT P x)) (terc x) =

```

$\text{intc } [] \text{ (just (PT } P \text{ x)) (inj}_2 \text{ x)}$
 $\text{(lemTrTerminateBind } c_0 \text{ P x)}$

--@END

$\text{unitSeqR}\infty : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c_0)$
 $\rightarrow (P \gg=\infty \text{ terminate}) \sqsubseteq\infty P$
 $\text{unitSeqR}\infty P \text{ l m q} = \text{unitSeqR (forcep P) l m q}$

mutual

$\text{unitSeq}_2\text{r} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (P : \text{Process} \infty \{lu\} c_0)$
 $\rightarrow P \sqsubseteq (P \gg= \text{ terminate})$
 $\text{unitSeq}_2\text{r } \{lu\} \{c_0\} (\text{terminate } x) \text{ l m } x_1 = x_1$
 $\text{unitSeq}_2\text{r } \{lu\} \{c_0\} (\text{node } x) \text{ l m (tnode tr)} = \text{tnode (unitSeq}_2\text{r}_+ x \text{ l m tr)}$

$\text{unitSeq}_2\text{r}_+ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (P : \text{Process}+ \infty \{lu\} c_0)$
 $\rightarrow P \sqsubseteq+ (P \gg=+ \text{ terminate})$
 $\text{unitSeq}_2\text{r}_+ P .[] .\text{nothing (empty } \{P = .(P \gg=+ \text{ terminate})\}) = \text{empty}$
 $\text{unitSeq}_2\text{r}_+ \{lu\} P .(\text{Lab } P \text{ x} :: l) \text{ m (extc } \{P = .(P \gg=+ \text{ terminate})\} \text{ l .m } x \text{ x}_1) = \text{extc l m}$
 $\text{unitSeq}_2\text{r}_+ \{lu\} \{c_0\} P \text{ l m (intc } \{P = .(-\gg=+- \{\infty\} \{lu\} \{c_0\} \{c_0\} P \text{ terminate})\} .\text{l .m (intc l m}$

$\text{unitSeq}_2\text{r}_+ \{lu\} \{c_0\} P \text{ l m (intc } \{P = .(-\gg=+- \{\infty\} \{lu\} \{c_0\} \{c_0\} P \text{ terminate})\} .\text{l .m (intc l m}$
 let

$s : \text{Set}$
 $s = \text{Tr } \{lu\} \{c_0\} \text{ l m (forcep (PI } (-\gg=+- \{\infty\} \{lu\} \{c_0\} \{c_0\} P \text{ terminate}))$

$\text{traux} : \text{Tr } \{lu\} \{c_0\} \text{ l m (terminate (PT } P \text{ y))}$
 $\text{traux} = x_1$

in $\text{lemtr}+\text{trterminate } c_0 \text{ m } P \text{ l y traux}$
 $\text{unitSeq}_2\text{r}_+ \{lu\} \{c_0\} P .[] .(\text{just (PT (P } \gg=+ \text{ terminate) x)) (terc } \{P = .(P \gg=+ \text{ terminate)$

$\text{unitSeq}_2\text{r}\infty : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (P : \text{Process}\infty \infty \{lu\} c_0)$
 $\rightarrow P \sqsubseteq\infty (P \gg=\infty \text{ terminate})$
 $\text{unitSeq}_2\text{r}\infty \{lu\} \{c_0\} P \text{ l m } x = \text{unitSeq}_2\text{r (forcep P } \{\infty\}) \text{ l m } x$

--@BEGIN@unitSeqREq

```

≡unitSeq₂ : {lu : LUniv}{c₀ c₁ : Choice} (P : Process ∞ {lu} c₀)
  → P ≡ (P >>= terminate)
≡unitSeq₂ {lu} {c₀} {c₁} P = (unitSeq₂r P) , (unitSeqR P)

```

```
--@END
```

mutual

```

assSeq : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → ((P >>= Q) >>= R) ⊆ (P >>= (λ x → Q x >>= R))
assSeq {lu} {c₀} {c₁} {c₂} (terminate x) Q R l m q = q
assSeq {lu} {c₀} {c₁} {c₂} (node x) Q R l m (tnode q) = tnode (assSeq₁₋₃₊ x Q R l m q)

```

```
--@BEGIN@assSeq
```

```

assSeq₁₋₃₊ : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process+ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → ((P >>=+ Q) >>=+ R)
  ⊆+ (P >>=+ (λ x → Q x >>= R))
assSeq₁₋₃₊ P Q R .[] .nothing empty = empty
assSeq₁₋₃₊ P Q R .(Lab P x :: l) m (extc l .m x x₁)
  = extc l m x (assSeq+pp P Q R l x m x₁)
assSeq₁₋₃₊ P Q R l m (intc .l .m (inj₁ x) x₁)
  = intc l m (inj₁ (inj₁ x))
  (assSeq∞pp (PI P x) Q R l m x₁)
assSeq₁₋₃₊ P Q R l m (intc .l .m (inj₂ y) x₁)
  = intc l m (inj₁ (inj₂ y))
  (assPT+pp P Q R y l m x₁)
assSeq₁₋₃₊ {lu} {c₀} {c₁} {c₂} P Q R .[]
  .(just (PT (P >>=+ (λ x → Q x >>= R)) x))
  (terc x) = intc [] (just (PT (λ x → Q x >>= R)) x)
  P (λ x₁ → λ x₂ → (Q x₁) R) x)
  (inj₂ x) (lemTrTerminateBind' c₀ c₁ c₂ P Q R x)

```

```
--@END
```

```

assSeq∞pp : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process∞ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → ((P >>=∞ Q) >>=∞ R) ⊑∞ (P >>=∞ (λ x → Q x >>= R))
assSeq∞pp {lu}{c₀} {c₁} {c₂} P Q R l m q = assSeq (forcep P) Q R l m q

```

```

assPT+pp : {lu : LUniv}{c₀ c₁ c₂ : Choice}(P : Process+ ∞ {lu} c₀)(Q : ChoiceSet c₀ → P
  → (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c₂))
  → (x : Tr∞ l m (PI (P >>=+ (λ x → Q x >>= R)) (inj₂ y)))
  → Tr∞ l m (PI (P >>=+ Q) (inj₂ y) >>=∞ R)
assPT+pp {lu} {c₀} {c₁} {c₂} P Q R y l m tr = tr

```

```

assSeq+pp : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process+ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → (l : List (Label lu))
  → (x : ChoiceSet (E P))
  → (m : Maybe (ChoiceSet c₂))
  → (x₁ : Tr∞ l m (⊔=>∞ (PE P x) (λ x₂ → ⊔=>∞ (Q x₂) R)))
  → Tr∞ l m (⊔=>∞ (⊔=>∞ (PE P x) Q) R)
assSeq+pp P Q R l x m tr = assSeq (forcep (PE P x)) Q R l m tr

```

mutual

```

assSeqr : {lu : LUniv}{c₀ c₁ c₂ : Choice} (P : Process ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  → (P >>= (λ x → Q x >>= R)) ⊑ ((P >>= Q) >>= R)
assSeqr (terminate x) Q R l m q = q
assSeqr (node x) Q R l m (tnode q) = tnode (assSeq₁₋₃+r x Q R l m q)

```

```

assSeq1-3+r : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P  >>=+ (λ x → Q x >>= R))  ⊑+ ((P >>=+ Q) >>=+ R)
assSeq1-3+r P Q R .[] .nothing empty = empty
assSeq1-3+r P Q R .(Lab P x :: l) m (extc l m x x1) = extc l m x (assSeq+ppr P Q R l x m x1)
assSeq1-3+r P Q R l m (intc .l m (inj1 (inj1 x)) x1) = intc l m (inj1 x) (assSeq∞ppr (PI P x) Q R l m x1)
assSeq1-3+r P Q R l m (intc .l m (inj1 (inj2 y)) x1) = intc l m (inj2 y) (assPT+ppr P Q R y l m x1)
assSeq1-3+r P Q R l m (intc .l m (inj2 ()) x1)
assSeq1-3+r P Q R .[] .(just (PT ((P >>=+ Q) >>=+ R) _)) (terc ())

```

```

assSeq+ppr : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (l : List (Label lu))
  → (x : ChoiceSet (E P))
  → (m : Maybe (ChoiceSet c2))
  → (x1 : Tr∞ l m (_>>=∞_ (_>>=∞_ (PE P x) Q) R))
  → Tr∞ l m (_>>=∞_ (PE P x) (λ x2 → _>>=_ (Q x2) R))
assSeq+ppr P Q R l x m x1 = assSeqr (forcep (PE P x)) Q R l m x1

```

```

assSeq∞ppr : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P  >>=∞ (λ x → Q x >>= R))  ⊑∞ ((P >>=∞ Q) >>=∞ R)
assSeq∞ppr {lu} {c0} {c1} {c2} P Q R l m q =      assSeqr (forcep P) Q R l m q

```

```

assPT+ppr : {lu : LUniv} {c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0) (Q : ChoiceSet c0 → Process
  → (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c2))
  → (x : Tr∞ l m (PI (P >>=+ Q) (inj2 y) >>=∞ R))
  →      Tr∞ l m (PI (P >>=+ (λ x → Q x >>= R)) (inj2 y))
assPT+ppr {lu} {c0} {c1} {c2} P Q R y l m tr = tr

```

```

--@BEGIN@assSeqQ

≡assSeq : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ {lu} c0)
          (Q : ChoiceSet c0 → Process ∞ {lu} c1)
          → (R : ChoiceSet c1 → Process ∞ {lu} c2)
          → ((P >>= Q) >>= R) ≡
              (P >>= (λ x → Q x >>= R))
≡assSeq P Q R = (assSeq P Q R) , (assSeqr P Q R)

--@END

```

A.79 ProofLawsSeqTheoOnly.agda

```

--@PREFIX@mainProofLawsSeqTheoOnly

```

```

module ProofLawsSeqTheoOnly where

```

```

open import process
open import Size
open import Level
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.Product
open import Data.Fin
open import Data.List
open import Data.Sum
open import labelUniv
open import dataAuxFunction
open import externalChoice
open import sequentialCompositionRev
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import primitiveProcess
open import traceEquivalence
open import Data.Product

```

```

lemTrTerminateBind : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(x : ChoiceSet (T P))
  → Tr∞ [] (just (PT P x)) (PI (P >>=+ terminate) (inj₂ x))
lemTrTerminateBind c P x = ter (PT P x)

```

```

lemTrTerminateBind' : {lu : LUniv}{c₀ c₁ c₂ : Choice}
  (P : Process+ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ c₁)
  (R : ChoiceSet c₁ → Process ∞ c₂)
  (x : Fin 0)
  → Tr∞ [] (just (PT (λ x₁ → _>>=+ P (λ x₁ → _>>=_ (Q x₁) R)) x))(PI (λ x₁ → _>>=_ (Q x₁) R)) x)
lemTrTerminateBind' c P Q R x q ()

```

```

--@BEGIN@stopSeq

```

```

stopSeq : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
  (P : ChoiceSet c₀ → Process ∞ {lu} c₀)
  → (STOP c₀ >>= P) ⊆ STOP c₀
stopSeq a P .[] .nothing (tnode empty) = tnode empty
stopSeq a P .(efq _ :: l) m (tnode (extc l .m () x₁))
stopSeq a P l m (tnode (intc l .m () x₁))
stopSeq a P .[] .(just (efq _)) (tnode (terc ()))

```

```

--@END

```

```

stopSeqr : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
  (P : ChoiceSet c₀ → Process ∞ {lu} c₀)
  → STOP c₀ ⊆ (STOP c₀ >>= P)
stopSeqr a P .[] .nothing (tnode empty) = tnode empty
stopSeqr a P .(efq _ :: l) m (tnode (extc l .m () x₁))
stopSeqr a P l m (tnode (intc l .m (inj₁ ()) x₁))
stopSeqr a P l m (tnode (intc l .m (inj₂ ()) x₁))
stopSeqr a P .[] .(just (PT (process+ (fin 0) efq efq
  (fin 0) efq (fin 0) efq "STOP"
  >>=+ P) _)) (tnode (terc ()))

```

--@BEGIN@stopSeqEq

$\equiv \text{stopSeq} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_0)$
 $\rightarrow \text{STOP } c_0 \{lu\} \equiv (\text{STOP } c_0 \{lu\} \gg= P)$

--@END

$\equiv \text{stopSeq } a P = (\text{stopSeqr } a P) , (\text{stopSeq } a P)$

--@BEGIN@unitSeqL

$\text{unitSeqL} : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1)$
 $\rightarrow (\text{terminate } a \gg= P) \sqsubseteq P \ a$

$\text{unitSeqL } a P \ l \ m \ q = q$

--@END

$\text{unitSeqLr} : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} (a : \text{ChoiceSet } c_0) (P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1)$
 $\rightarrow P \ a \sqsubseteq (\text{terminate } a \gg= P)$

$\text{unitSeqLr } \{lu\} \{c_0\} \{c_1\} a P \ l \ m \ q = q$

--@BEGIN@unitSeqLEq

$\equiv \text{unitSeq} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} (a : \text{ChoiceSet } c_0)$
 $(P : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_0)$
 $\rightarrow P \ a \equiv (\text{terminate } a \gg= P)$

$\equiv \text{unitSeq } a P = (\text{unitSeqL } a P) , \text{unitSeqLr } a P$

--@END

```

lemTrTerminateBind'' : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c) (x : Fin 0)
  → Tr+ {lu} [] (just (PT (P >>=+ terminate) x)) P
lemTrTerminateBind'' c P ()

```

```

lemtr+trterminate : {lu : LUniv}(c0 : Choice) → (m : Maybe (ChoiceSet c0)) → (P : Process+ ∞ {lu} c0)
  → (y : ChoiceSet (T P)) → (traux : Tr {lu} {c0} l m (terminate (PT P y))) → Tr {lu} {c0} l m (traux y)
lemtr+trterminate c0 .(just (PT P y)) P .[] y (ter .(PT P y)) = terc y
lemtr+trterminate c0 .nothing P .[] y (empty .(PT P y)) = empty

```

mutual

```

unitSeqR : {lu : LUniv}{c0 : Choice} (P : Process ∞ {lu} c0)
  → (P >>= terminate) ⊆ P
unitSeqR (terminate x) l m q = q
unitSeqR (node x) l m (tnode q) = tnode (unitSeqR+ x l m q)

```

--@BEGIN@unitSeqR

```

unitSeqR+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)
  → (P >>=+ terminate) ⊆+ P
unitSeqR+ P .[] .nothing empty
  = empty
unitSeqR+ P .(Lab P x :: l) m (extc l .m x x1)
  = extc l m x (unitSeqR∞ (PE P x) l m x1)
unitSeqR+ P l m (intc .l .m x x1)
  = intc l m (inj1 x) (unitSeqR∞ (PI P x) l m x1)
unitSeqR+ {lu} {c0} P .[] .(just (PT P x)) (terc x) =
  intc [] (just (PT P x)) (inj2 x)
  (lemTrTerminateBind c0 P x)

```

--@END

unitSeqR ∞ : {lu : LUniv}{c₀ : Choice} (P : Process ∞ ∞ {lu} c₀)
 \rightarrow (P $\gg=\infty$ terminate) $\sqsubseteq\infty$ P
 unitSeqR ∞ P l m q = unitSeqR (forcep P) l m q

mutual

unitSeq₂r : {lu : LUniv}{c₀ : Choice} (P : Process ∞ {lu} c₀)
 \rightarrow P \sqsubseteq (P $\gg=$ terminate)
 unitSeq₂r {lu} {c₀} (terminate x) l m x₁ = x₁
 unitSeq₂r {lu} {c₀} (node x) l m (tnode tr) = tnode (unitSeq₂r₊ x l m tr)

unitSeq₂r₊ : {lu : LUniv}{c₀ : Choice} (P : Process₊ ∞ {lu} c₀)
 \rightarrow P $\sqsubseteq+$ (P $\gg=+$ terminate)

unitSeq₂r₊ P .[] .nothing (empty {P = .(P $\gg=+$ terminate)}) = empty

unitSeq₂r₊ {lu} P .(Lab P x :: l) m (extc {P = .(P $\gg=+$ terminate)} l .m x x₁) = extc l m

unitSeq₂r₊ {lu} {c₀} P l m (intc {P = .(_ $\gg=+$ _ { ∞ } {lu} {c₀} {c₀} P terminate)} .l .m (intc l m

unitSeq₂r₊ {lu}{c₀} P l m (intc {P = .(_ $\gg=+$ _ { ∞ } {lu}{c₀} {c₀} P terminate)} .l .m (intc l m

let

s : Set

s = Tr {lu} {c₀} l m (forcep (PI (_ $\gg=+$ _ { ∞ } {lu} {c₀} {c₀} P terminate)

traux : Tr {lu} {c₀} l m (terminate (PT P y))

traux = x₁

in lemtr+trterminate c₀ m P l y traux

unitSeq₂r₊ {lu} {c₀} P .[] .(just (PT (P $\gg=+$ terminate) x)) (terc {P = .(P $\gg=+$ terminate)

unitSeq₂r ∞ : {lu : LUniv}{c₀ : Choice} (P : Process ∞ ∞ {lu} c₀)
 \rightarrow P $\sqsubseteq\infty$ (P $\gg=\infty$ terminate)

unitSeq₂r ∞ {lu} {c₀} P l m x = unitSeq₂r (forcep P { ∞ }) l m x

--@BEGIN@unitSeqREq

\equiv unitSeq₂ : {lu : LUniv}{c₀ c₁ : Choice} (P : Process ∞ {lu} c₀)
 \rightarrow P \equiv (P $\gg=$ terminate)

```
≡ unitSeq2 {lu} {c0} {c1} P = (unitSeq2r P) , (unitSeqR P)
```

```
--@END
```

```
mutual
```

```
assSeq : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process ∞ {lu} c0)
      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → ((P >>= Q) >>= R) ⊆ (P >>= (λ x → Q x >>= R))
assSeq {lu} {c0} {c1} {c2} (terminate x) Q R l m q = q
assSeq {lu} {c0} {c1} {c2} (node x) Q R l m (tnode q) = tnode (assSeq1-3+ x Q R l m q)
```

```
--@BEGIN@assSeq
```

```
assSeq1-3+ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → ((P >>=+ Q) >>=+ R)
      ⊆+ (P >>=+ (λ x → Q x >>= R))
assSeq1-3+ P Q R .[] .nothing empty = empty
assSeq1-3+ P Q R .(Lab P x :: l) m (extc l .m x x1)
    = extc l m x (assSeq+pp P Q R l x m x1)
assSeq1-3+ P Q R l m (intc .l .m (inj1 x) x1)
    = intc l m (inj1 (inj1 x))
      (assSeq∞pp (PI P x) Q R l m x1)
assSeq1-3+ P Q R l m (intc .l .m (inj2 y) x1)
    = intc l m (inj1 (inj2 y))
      (assPT+pp P Q R y l m x1)
assSeq1-3+ {lu} {c0} {c1} {c2} P Q R .[]
    .(just (PT (P >>=+ (λ x → Q x >>= R)) x))
    (terc x) = intc [] (just (PT (λ x → Q x >>= R)) x)
      (inj2 x) (lemTrTerminateBind' c0 c1 c2 P Q R x)
```

```
--@END
```

```
assSeq∞pp : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
```

$$\begin{aligned}
 & (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) \\
 & \rightarrow ((P \gg=\infty Q) \gg=\infty R) \sqsubseteq_\infty (P \gg=\infty (\lambda x \rightarrow Q x \gg= R)) \\
 \text{assSeq}\infty\text{pp } \{lu\} \{c_0\} \{c_1\} \{c_2\} P Q R l m q = & \text{assSeq } (\text{forcep } P) Q R l m q
 \end{aligned}$$

$$\begin{aligned}
 \text{assPT+pp} : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c_0) (Q : \text{ChoiceSet } c_0 \rightarrow P) \rightarrow & \\
 & (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) \\
 & \rightarrow (y : \text{ChoiceSet } (\text{T } P)) \\
 & \rightarrow (l : \text{List } (\text{Label } lu)) \\
 & \rightarrow (m : \text{Maybe } (\text{ChoiceSet } c_2)) \\
 & \rightarrow (x : \text{Tr}_\infty l m (\text{PI } (P \gg=+ (\lambda x \rightarrow Q x \gg= R)) (\text{inj}_2 y))) \\
 & \rightarrow \text{Tr}_\infty l m (\text{PI } (P \gg=+ Q) (\text{inj}_2 y) \gg=\infty R) \\
 \text{assPT+pp } \{lu\} \{c_0\} \{c_1\} \{c_2\} P Q R y l m tr = & tr
 \end{aligned}$$

$$\begin{aligned}
 \text{assSeq+pp} : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c_0) & \\
 (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) & \\
 (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) & \\
 \rightarrow (l : \text{List } (\text{Label } lu)) & \\
 \rightarrow (x : \text{ChoiceSet } (\text{E } P)) & \\
 \rightarrow (m : \text{Maybe } (\text{ChoiceSet } c_2)) & \\
 \rightarrow (x_1 : \text{Tr}_\infty l m (_ \gg=\infty _ (\text{PE } P x) (\lambda x_2 \rightarrow _ \gg= _ (Q x_2) R))) & \\
 \rightarrow \text{Tr}_\infty l m (_ \gg=\infty _ (_ \gg=\infty _ (\text{PE } P x) Q) R) & \\
 \text{assSeq+pp } P Q R l x m tr = & \text{assSeq } (\text{forcep } (\text{PE } P x)) Q R l m tr
 \end{aligned}$$

mutual

$$\begin{aligned}
 \text{assSeqr} : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process } \infty \{lu\} c_0) & \\
 (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) & \\
 (R : \text{ChoiceSet } c_1 \rightarrow \text{Process } \infty \{lu\} c_2) & \\
 \rightarrow (P \gg= (\lambda x \rightarrow Q x \gg= R)) \sqsubseteq ((P \gg= Q) \gg= R) & \\
 \text{assSeqr } (\text{terminate } x) Q R l m q = & q \\
 \text{assSeqr } (\text{node } x) Q R l m (\text{tnode } q) = & \text{tnode } (\text{assSeq}_{1-3+r} x Q R l m q)
 \end{aligned}$$

$$\begin{aligned}
 \text{assSeq}_{1-3+r} : \{lu : \text{LUniv}\} \{c_0 c_1 c_2 : \text{Choice}\} (P : \text{Process+ } \infty \{lu\} c_0) & \\
 (Q : \text{ChoiceSet } c_0 \rightarrow \text{Process } \infty \{lu\} c_1) &
 \end{aligned}$$

```

      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → (P  >>=+ ( λ x → Q x >>= R ))  ⊆+ ((P >>=+ Q) >>=+ R)
assSeq1-3+r P Q R .[] .nothing empty = empty
assSeq1-3+r P Q R .(Lab P x :: l) m (extc l m x x1) = extc l m x (assSeq+ppr P Q R l x m x1)
assSeq1-3+r P Q R l m (intc .l m (inj1 (inj1 x)) x1) = intc l m (inj1 x) (assSeq∞ppr (PI P x) Q R l m x1)
assSeq1-3+r P Q R l m (intc .l m (inj1 (inj2 y)) x1) = intc l m (inj2 y) (assPT+ppr P Q R y l m x1)
assSeq1-3+r P Q R l m (intc .l m (inj2 ()) x1)
assSeq1-3+r P Q R .[] .(just (PT ((P >>=+ Q) >>=+ R) _)) (terc ())

```

```

assSeq+ppr : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → (l      : List (Label lu))
      → (x      : ChoiceSet (E P))
      → (m      : Maybe (ChoiceSet c2))
      → (x1 : Tr∞ l m (_>>=∞_ (_>>=∞_ (PE P x) Q) R))
      → Tr∞ l m (_>>=∞_ (PE P x) (λ x2 → _>>=_ (Q x2) R))
assSeq+ppr P Q R l x m x1 = assSeqr (forcep (PE P x)) Q R l m x1

```

```

assSeq∞ppr : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → (P  >>=∞ ( λ x → Q x >>= R ))  ⊆∞ ((P >>=∞ Q) >>=∞ R)
assSeq∞ppr {lu} {c0} {c1} {c2} P Q R l m q =      assSeqr (forcep P) Q R l m q

```

```

assPT+ppr : {lu : LUniv} {c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0) (Q : ChoiceSet c0 → Process
      → (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → (y : ChoiceSet (T P))
      → (l : List (Label lu))
      → (m : Maybe (ChoiceSet c2))
      → (x : Tr∞ l m (PI (P >>=+ Q) (inj2 y) >>=∞ R))
      → Tr∞ l m (PI (P >>=+ (λ x → Q x >>= R)) (inj2 y))
assPT+ppr {lu} {c0} {c1} {c2} P Q R y l m tr = tr

```

```
--@BEGIN@assSeqeQ
```

```
≡assSeq : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process ∞ {lu} c0)
```

```

      (Q : ChoiceSet c0 → Process ∞ {lu} c1)
    → (R : ChoiceSet c1 → Process ∞ {lu} c2)
    → ((P >>= Q) >>= R) ≡
      (P >>= (λ x → Q x >>= R))
≡ assSeq P Q R = (assSeq P Q R) , (assSeqr P Q R)

```

```
--@END
```

A.80 proofMonadicLaw.agda

```
--@PREFIX@mainproofMonadicLaw
```

```
module proofMonadicLaw where
```

```

open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.Product
open import Data.Fin
open import Data.List
open import Data.Sum
open import dataAuxFunction
open import sequentialCompositionRev
open import TraceWithoutSize
open import RefWithoutSize
open import traceEquivalence
open import labelUniv

```

```
--@BEGIN@lemTrTerminateBindDef
```

```

lemTrTerminateBind : {lu : LUniv} {c : Choice} (P : Process+ ∞ {lu} c) (x : ChoiceSet (T P))
  → Tr∞ [] (just (PT P x)) (PI (P >>=+ terminate) (inj2 x))
lemTrTerminateBind c P x = ter (PT P x)

```

```

lemTrTerminateBind'' : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c) (x : Fin 0)
    → Tr+ [] (just (PT (P >>=+ terminate) x)) P
lemTrTerminateBind'' c P ()

```

```
--@END
```

```
--@BEGIN@lemTrTerminateBindoneDef
```

```

lemTrTerminateBind' : {lu : LUniv}(c0 c1 c2 : Choice)
    (P : Process+ ∞ {lu} c0)
    (Q : ChoiceSet c0 → Process ∞ {lu} c1)
    (R : ChoiceSet c1 → Process ∞ {lu} c2)
    (x : Fin 0)
    → Tr∞ [] (just (PT (P >>=+ P (λ x1 → >>=_ (Q x1) R)) x))(PI (P >>=+
lemTrTerminateBind' c P Q R x q ()

```

```
--@END
```

```
--@BEGIN@lemTrTerminateBindtwoDef
```

```

lemTrTerminateBind''' : {lu : LUniv}(c0 c1 c2 : Choice)
    (P : Process+ ∞ {lu} c0)
    (Q : ChoiceSet c0 → Process ∞ {lu} c1)
    (R : ChoiceSet c1 → Process ∞ {lu} c2)
    (x : Fin 0)
    → Tr+ [] (just (PT ((P >>=+ Q) >>=+ R) x))(P >>=+ (λ x1 → (Q x1) >>
lemTrTerminateBind''' c0 c1 c2 P Q R ()

```

```
--@END
```

```
--@BEGIN@monadicLawoneDef
```

```

monadicLaw1 : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
    (P : ChoiceSet c0 → Process ∞ {lu} c1)
    → (terminate a >>= P) ⊆ P a
monadicLaw1 a P l m q = q

```

```
--@END
```

```
--@BEGIN@monadicLawonerDef
```

```
monadicLaw1R : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → P a ⊆ (terminate a >>= P)
monadicLaw1R {c0} {c1} a P l m q = q
```

```
--@END
```

```
--@BEGIN@monadicLawoneEqDef
```

```
≡monadicLaw1 : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (P a) ≡ (terminate a >>= P)
≡monadicLaw1 {c0} {c1} a P = (monadicLaw1 a P) , (monadicLaw1R a P)
```

```
--@END
```

```
mutual
```

```
--@BEGIN@monadicLawtwoDef
```

```
monadicLaw2 : {lu : LUniv}{c0 : Choice} (P : Process ∞ {lu} c0)
  → (P >>= terminate) ⊆ P
monadicLaw2 {c0} (terminate x) l m q = q
monadicLaw2 {c0} (node x) l m (tnode q) = tnode (monadicLaw2+ x l m q)
```

```
--@END
```

```
--@BEGIN@monadicLawtowPlusDef
```

```
monadicLaw2+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)
  → (P >>=+ terminate) ⊆+ P
monadicLaw2+ P .[] .nothing empty = empty
monadicLaw2+ P .(Lab P x :: l) m (extc l .m x x1) =
  extc l m x
  (monadicLaw2∞ (PE P x) l m x1)
```

```

monadicLaw2+  P l m (intc .l .m x x1) =
    intc l m (inj1 x)
    (monadicLaw2∞ (PI P x) l m x1)
monadicLaw2+  {lu} {c0} P .[] .(just (PT P x)) (terc x) =
    intc [] (just (PT P x)) (inj2 x)
    (lemTrTerminateBind c0 P x)

--@END

--@BEGIN@monadicLawtowInfDef

monadicLaw2∞ : {lu : LUniv}{c0 : Choice} (P : Process∞ ∞ {lu} c0)
    → (P >>=∞ terminate) ⊆∞ P
monadicLaw2∞ {lu} {c0} P l m q = monadicLaw2 (forcep P) l m q

--@END

--@BEGIN@lemtrPlustrterminateDef

lemtr+trterminate : {lu : LUniv}(c0 : Choice) → (m : Maybe (ChoiceSet c0))
    → (P : Process+ ∞ {lu} c0) → (l : List (Label lu)) →
    (y : ChoiceSet (T P)) → (traux : Tr {lu} {c0} l m (terminate (PT P y))) →
lemtr+trterminate c0 .(just (PT P y)) P .[] y (ter .(PT P y)) = terc y
lemtr+trterminate c0 .nothing P .[] y (empty .(PT P y)) = empty

--@END

mutual
--@BEGIN@monadicLawtwoRDef

monadicLaw2R : {lu : LUniv}{c0 : Choice} (P : Process ∞ {lu} c0)
    → P ⊆ (P >>= terminate)
monadicLaw2R {lu} {c0} (terminate x) l m x1 = x1
monadicLaw2R {lu} {c0} (node x) l m (tnode tr) = tnode (monadicLaw2R+ x l m tr)

--@END

--@BEGIN@monadicLawtwoRPlusDef

monadicLaw2R+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)
    → P ⊆+ (P >>=+ terminate)

```

```

monadicLaw2R+ P .[] .nothing (empty {P = .(P >>=+ terminate)}) = empty
monadicLaw2R+ P .(Lab P x :: l) m (extc {P = .(P >>=+ terminate)} l .m x x1) =
    extc l m x (monadicLaw2R∞ (PE P x) l m x1)
monadicLaw2R+ {lu} {c0} P l m (intc {P = .(->>=+- {∞} {lu} {c0} {c0} P terminate)} l m x) =
    intc l m x (monadicLaw2R∞ (PI P x) l m x)
monadicLaw2R+ {lu} {c0} P l m (intc {P = .(->>=+- {∞} {lu} {c0} {c0} P terminate)} l m x) =
    intc l m x (monadicLaw2R∞ (PI P x) l m x)
let
  s : Set
  s = Tr {lu} {c0} l m (forcep (PI (->>=+- {∞} {lu} {c0} {c0} P terminate) l m x)
    traux : Tr {lu} {c0} l m (terminate (PT P y))
    traux = x1

in lemtr+trterminate c0 m P l y traux
monadicLaw2R+ {lu} {c0} P .[] .(just (PT (P >>=+ terminate) x)) (terc {P = .(P >>=+ terminate)} l m x)

--@END

--@BEGIN@monadicLawtwoInfDef

monadicLaw2R∞ : {lu : LUniv}{c0 : Choice} (P : Process∞ ∞ {lu} c0)
  → P ⊆∞ (P >>=∞ terminate)
monadicLaw2R∞ {lu} {c0} P l m x = monadicLaw2R (forcep P {∞}) l m x

--@END

--@BEGIN@monadicLawtwoEqDef

≡monadicLaw2 : {lu : LUniv}{c0 c1 : Choice} (P : Process ∞ {lu} c0)
  → P ≡ (P >>= terminate)
≡monadicLaw2 {lu} {c0} {c1} P = (monadicLaw2R P) , (monadicLaw2R P)

--@END

mutual

--@BEGIN@monadicLawthreeDef

```

○

○

```

monadicLaw3 : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>= Q) >>= R) ⊆ (P >>= (λ x → Q x >>= R))
monadicLaw3 (terminate x) Q R l m q = q
monadicLaw3 (node x) Q R l m (tnode q) = tnode (monadicLaw3+ x Q R l m q)

--@END

--@BEGIN@monadicLawthreePlusDef

monadicLaw3+ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>=+ Q) >>=+ R) ⊆+
    (P >>=+ (λ x → Q x >>= R))
monadicLaw3+ P Q R .[] .nothing empty = empty
monadicLaw3+ P Q R .(Lab P x :: l) m (extc l m x x1) =
  extc l m x
  (monadicLaw∞ P Q R l x m x1)
monadicLaw3+ P Q R l m (intc l m (inj1 x) x1) =
  intc l m (inj1 (inj1 x))
  (monadicLaw3∞ (PI P x) Q R l m x1)
monadicLaw3+ P Q R l m (intc l m (inj2 y) x1) =
  intc l m (inj1 (inj2 y))
  (monadPT+ P Q R y l m x1)
monadicLaw3+ P Q R .[] .(just (PT
  (P >>=+ (λ x → Q x >>= R)) x)) (terc x) = efq x

--@END

--@BEGIN@monadicLawthreeInfDef

monadicLaw3∞ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>=∞ Q) >>=∞ R) ⊆∞ (P >>=∞ (λ x → Q x >>= R))
monadicLaw3∞ {lu} {c0} {c1} {c2} P Q R l m q = monadicLaw3 (forcep P) Q R l m q

--@END

```

○

○

```
--@BEGIN@monadPTDef
```

```
monadPT+ : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  → (R      : ChoiceSet c1 → Process ∞ {lu} c2)
  → (y      : ChoiceSet (T P))
  → (l      : List (Label lu))
  → (m      : Maybe (ChoiceSet c2))
  → (x : Tr∞ l m (PI (P >>=+ (λ x → Q x >>= R)) (inj2 y)))
  →      Tr∞ l m (PI (P >>=+ Q) (inj2 y) >>=∞ R)
monadPT+ {lu} {c0} {c1} {c2} P Q R y l m tr = tr
```

```
--@END
```

```
--@BEGIN@monadDownSizeDef
```

```
monadDownSize : {lu : LUniv}(c0 c1 : Choice) (P : Process ∞ {lu} c0)(Q : ChoiceSet c0 →
  → (l      : List (Label lu) )
  → (m      : Maybe (ChoiceSet c1))
  → Tr l m (_>>=_ P Q)
  → Tr l m (_>>=_ P Q)
monadDownSize c0 c1 (terminate x) Q l m tr = tr
monadDownSize c0 c1 (node x) Q l m (tnode {l = .l} {x = .m} {P = .(_>>=+_ x Q)}) x1 =
```

```
--@END
```

```
--@BEGIN@monadDownSizePlusDef
```

```
monadDownSize+ : {lu : LUniv}(c0 c1 : Choice) (P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0
  → (l      : List (Label lu))
  → (m      : Maybe (ChoiceSet c1))
  → Tr+ l m (_>>=+_ P Q)
  → Tr+ l m (_>>=+_ P Q)
monadDownSize+ c0 c1 q P .[] .nothing (empty {P = .(_>>=+_ q P)}) = empty
monadDownSize+ c0 c1 q P .(Lab q x :: l) m (extc {P = .(_>>=+_ q P)} l .m x x1) = extc
monadDownSize+ c0 c1 q P l m (intc {P = .(_>>=+_ q P)} .l .m (inj1 x) x1) = intc l m (inj
monadDownSize+ c0 c1 q P l m (intc {P = .(_>>=+_ q P)} .l .m (inj2 y) x1) = intc l m (inj
monadDownSize+ c0 c1 q P .[] .(just (PT (_>>=+_ q P) x)) (terc {P = .(_>>=+_ q P)} x)
```

```
--@END
```

```
--@BEGIN@monadDownSizePTDef
```

```
monadDownSizePT : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0 → Process+ ∞ {lu} c1)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr∞ l m (PI (_>>=+_ P Q) (inj2 y))
  → Tr∞ l m (PI (_>>=+_ P Q) (inj2 y))
monadDownSizePT c0 c1 P Q y l m tr = tr
```

```
--@END
```

```
--@BEGIN@monadDownSizeInfDef
```

```
monadDownSize∞ : {lu : LUniv}{c0 c1 : Choice} (P : Process∞ ∞ {lu} c0)(Q : ChoiceSet c0 → Process∞ ∞ {lu} c1)
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr∞ l m (_>>=∞_ P Q)
  → Tr∞ l m (_>>=∞_ P Q)
monadDownSize∞ c0 c1 P Q l m tr = tr
```

```
--@END
```

```
--@BEGIN@monadicLawInfDef
```

```
monadicLaw∞ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process∞ ∞ {lu} c1)
  (R : ChoiceSet c1 → Process∞ ∞ {lu} c2)
  → (l : List (Label lu))
  → (x : ChoiceSet (E P))
  → (m : Maybe (ChoiceSet c2))
  → (x1 : Tr∞ l m (_>>=∞_ (PE P x) (λ x2 → _>>=_ (Q x2) R)))
  → Tr∞ l m (_>>=∞_ (_>>=∞_ (PE P x) Q) R)
monadicLaw∞ P Q R l x m tr = monadicLaw3 (forcep (PE P x)) Q R l m tr
```

```
--@END
```

mutual

--@BEGIN@monadicLawRThreeDef

```
monadicLaw3R : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P >>= (λ x → Q x >>= R)) ⊆ ((P >>= Q) >>= R)
```

```
monadicLaw3R {lu} {c0} {c1} {c2} (terminate x) Q R l m x1 = x1
```

```
monadicLaw3R {lu} {c0} {c1} {c2} (node x) Q R l m (tnode x1) = tnode (monadicLaw3R+ x1)
```

--@END

--@BEGIN@monadicLawRPlusThreeDef

```
monadicLaw3R+ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P >>=+ (λ x → Q x >>= R)) ⊆+ ((P >>=+ Q) >>=+ R)
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .[] .nothing (empty {P
= .(>>=+_ {∞} {lu} {c1} {c2} (>>=+_ {∞} {lu} {c0} {c1} P Q) R)) = empty
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .(Lab P x :: l) m (extc {P
= .(>>=+_ {∞} {lu} {c1} {c2} (>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m x x1)
= extc l m x (monadicLaw3R∞ (PE P x) Q R l m x1)
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
= .(>>=+_ {∞} {lu} {c1} {c2} (>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj1 (inj1
= intc l m (inj1 x) (monadicLaw3R∞ (PI P x) Q R l m x
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
= .(>>=+_ {∞} {lu} {c1} {c2} (>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj1 (inj2
= intc l m (inj2 y) (monadPT+' P Q R y l m x1)
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
= .(>>=+_ {∞} {lu} {c1} {c2} (>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj2 ())
```

```
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .[] .(just (PT (>>=+_ (P >>=+ Q) R) x))
(terc {P = .(>>=+_ (P >>=+ Q) R)} x)
= lemTrTerminateBind'' c0 c1 c2 P Q R x
```

--@END

--@BEGIN@monadicLawTreeInfDef

```
monadicLaw3R∞ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
```


○

○

```

      (R : ChoiceSet c1 → Process ∞ {lu} c2)
      → (P ≫=∞ (λ x → Q x ≫= R)) ⊆∞ ((P ≫=∞ Q) ≫=∞ R)
monadicLaw3R ∞ {lu} {c0} {c1} {c2} P Q R l m q = monadicLaw3R (forcep P) Q R l m q

--@END

--@BEGIN@monadPTplusDef

monadPT+' : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0 → Process
  → (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c2))
  → (x : Tr ∞ l m (≫=∞_ (PI (≫=+_ P Q) (inj2 y)) R))
  → Tr ∞ l m (PI (≫=+_ P (λ x → ≫=_ (Q x) R)) (inj2 y))
monadPT+' {lu} {c0} {c1} {c2} P Q R y l m tr = tr

--@END

--@BEGIN@monadicLawEqThreeDef

≡monadicLaw3 : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P ≫= Q) ≫= R) ≡ (P ≫= (λ x → Q x ≫= R))
≡monadicLaw3 {lu} {c0} {c1} {c2} P Q R = (monadicLaw3 P Q R) ,
  (monadicLaw3R P Q R)

--@END

```

A.81 proofMonadicLawTheoremsOnly.agda

```
--@PREFIX@mainproofMonadicLawTheoremsOnly
```

○

○

```
module proofMonadicLawTheoremsOnly where
```

```
open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.Product
open import Data.Fin
open import Data.List
open import Data.Sum
open import dataAuxFunction
open import sequentialCompositionRev
open import TraceWithoutSize
open import RefWithoutSize
open import traceEquivalence
open import labelUniv
```

```
--@BEGIN@lemTrTerminateBindDef
```

```
lemTrTerminateBind : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)(x : ChoiceSet (T P))
  → Tr∞ [] (just (PT P x)) (PI (P >>=+ terminate) (inj₂ x))
lemTrTerminateBind c P x = ter (PT P x)
```

```
lemTrTerminateBind'' : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c) (x : Fin 0)
  → Tr+ [] (just (PT (P >>=+ terminate) x)) P
lemTrTerminateBind'' c P ()
```

```
--@END
```

```
--@BEGIN@lemTrTerminateBindoneDef
```

```
lemTrTerminateBind' : {lu : LUniv}{c₀ c₁ c₂ : Choice}
  (P : Process+ ∞ {lu} c₀)
  (Q : ChoiceSet c₀ → Process ∞ {lu} c₁)
  (R : ChoiceSet c₁ → Process ∞ {lu} c₂)
  (x : Fin 0)
  → Tr∞ [] (just (PT (P >>=+ P (λ x₁ → P >>= (Q x₁) R)) x)) (PI
lemTrTerminateBind' c P Q R x q ()
```

```
--@END
```

```
--@BEGIN@lemTrTerminateBindtwoDef
```

```
lemTrTerminateBind''' : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  (x : Fin 0)
    → Tr+ [] (just (PT ((P >>=+ Q) >>=+ R) x))(P >>=+ (λ x1 → (Q x1) >>
lemTrTerminateBind''' c0 c1 c2 P Q R ()
```

```
--@END
```

```
--@BEGIN@monadicLawoneDef
```

```
monadicLaw1 : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (terminate a >>= P) ⊆ P a
```

```
--@END
```

```
monadicLaw1 a P l m q = q
```

```
--@BEGIN@monadicLawonerDef
```

```
monadicLaw1R : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → P a ⊆ (terminate a >>= P)
```

```
--@END
```

```
monadicLaw1R {c0} {c1} a P l m q = q
```

```
--@BEGIN@monadicLawoneEqDef
```

```

≡monadicLaw1 : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (P : ChoiceSet c0 → Process ∞ {lu} c1)
  → (P a) ≡ (terminate a >>= P)

--@END
≡monadicLaw1 {c0} {c1} a P = (monadicLaw1 a P) , (monadicLaw1R a P)

mutual
--@BEGIN@monadicLawtwoDef

monadicLaw2 : {lu : LUniv}{c0 : Choice}
  (P : Process ∞ {lu} c0)
  → (P >>= terminate) ⊆ P

--@END
monadicLaw2 {c0} (terminate x) l m q = q
monadicLaw2 {c0} (node x) l m (tnode q) = tnode (monadicLaw2+ x l m q)

--@BEGIN@monadicLawtowPlusDef

monadicLaw2+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)
  → (P >>=+ terminate) ⊆+ P

--@END
monadicLaw2+ P .[] .nothing empty = empty
monadicLaw2+ P .(Lab P x :: l) m (extc l .m x x1) =
  extc l m x
  (monadicLaw2 ∞ (PE P x) l m x1)
monadicLaw2+ P l m (intc l .m x x1) =
  intc l m (inj1 x)
  (monadicLaw2 ∞ (PI P x) l m x1)

```

○

```

monadicLaw2+ {lu} {c0} P .[] .(just (PT P x)) (terc x) =
  intc [] (just (PT P x)) (inj2 x)
  (lemTrTerminateBind c0 P x)

```

```
--@BEGIN@monadicLawtowInfDef
```

```

monadicLaw2∞ : {lu : LUniv}{c0 : Choice} (P : Process∞ ∞ {lu} c0)
  → (P >>=∞ terminate) ⊑∞ P

```

```
--@END
```

```

monadicLaw2∞ {lu} {c0} P l m q = monadicLaw2 (forcep P) l m q

```

```
--@BEGIN@lemtrPlustrterminateDef
```

```

lemtr+trterminate : {lu : LUniv}(c0 : Choice) → (m : Maybe (ChoiceSet c0))
  → (P : Process+ ∞ {lu} c0) → (l : List (Label lu)) →
    (y : ChoiceSet (T P)) → (traux : Tr {lu} {c0} l m (terminate (PT P y))) →
lemtr+trterminate c0 .(just (PT P y)) P .[] y (ter .(PT P y)) = terc y
lemtr+trterminate c0 .nothing P .[] y (empty .(PT P y)) = empty

```

```
--@END
```

```
mutual
```

```
--@BEGIN@monadicLawtwoRDef
```

```

monadicLaw2R : {lu : LUniv}{c0 : Choice} (P : Process ∞ {lu} c0)
  → P ⊑ (P >>= terminate)

```

```
--@END
```

```

monadicLaw2R {lu} {c0} (terminate x) l m x1 = x1
monadicLaw2R {lu} {c0} (node x) l m (tnode tr) = tnode (monadicLaw2R+ x l m tr)

```

```
--@BEGIN@monadicLawtwoRPlusDef
```

```

monadicLaw2R+ : {lu : LUniv}{c0 : Choice} (P : Process+ ∞ {lu} c0)

```

○

$$\rightarrow P \sqsubseteq_+ (P \gg=+ \text{terminate})$$

--@END

```

monadicLaw2R+ P .[] .nothing (empty {P = .(P >>=+ terminate)}) = empty
monadicLaw2R+ P .(Lab P x :: l) m (extc {P = .(P >>=+ terminate)} l .m x x1) =
    extc l m x (monadicLaw2R∞ (PE P x) l m x1)
monadicLaw2R+ {lu} {c0} P l m (intc {P = .(->>=+- {∞} {lu} {c0} {c0} P terminate)} l m x) =
    intc l m x (monadicLaw2R∞ (PI P x) l m x1)
monadicLaw2R+ {lu} {c0} P l m (intc {P = .(->>=+- {∞} {lu} {c0} {c0} P terminate)} l m x) =
    intc l m x (monadicLaw2R∞ (PI P x) l m x1)
let
  s : Set
  s = Tr {lu} {c0} l m (forcep (PI (->>=+- {∞} {lu} {c0} {c0} P terminate) x))
  traux : Tr {lu} {c0} l m (terminate (PT P y))
  traux = x1

```

```

in lemtr+trterminate c0 m P l y traux
monadicLaw2R+ {lu} {c0} P .[] .(just (PT (P >>=+ terminate) x)) (terc {P = .(P >>=+ terminate) x})

```

--@BEGIN@monadicLawtwoInfDef

```

monadicLaw2R∞ : {lu : LUniv} {c0 : Choice} (P : Process ∞ {lu} c0)
  → P ⊆∞ (P >>=∞ terminate)

```

--@END

```

monadicLaw2R∞ {lu} {c0} P l m x = monadicLaw2R (forcep P {∞}) l m x

```

--@BEGIN@monadicLawtwoEqDef

```

≡monadicLaw2 : {lu : LUniv} {c0 c1 : Choice}
  (P : Process ∞ {lu} c0)
  → P ≡ (P >>= terminate)

```

--@END

```

≡monadicLaw2 {lu} {c0} {c1} P = (monadicLaw2R P) , (monadicLaw2 P)

```

mutual

```
--@BEGIN@monadicLawthreeDef
```

```
monadicLaw3 : {lu : LUniv}{c0 c1 c2 : Choice}
  (P : Process ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>= Q) >>= R) ⊆ (P >>= (λ x → Q x >>= R))
```

```
--@END
```

```
monadicLaw3 (terminate x) Q R l m q = q
monadicLaw3 (node x) Q R l m (tnode q) = tnode (monadicLaw3+ x Q R l m q)
```

```
--@BEGIN@monadicLawthreePlusDef
```

```
monadicLaw3+ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>=+ Q) >>=+ R) ⊆+
    (P >>=+ (λ x → Q x >>= R))
```

```
--@END
```

```
monadicLaw3+ P Q R .[] .nothing empty = empty
monadicLaw3+ P Q R .(Lab P x :: l) m (extc l .m x x1) =
  extc l m x
  (monadicLaw∞ P Q R l x m x1)
monadicLaw3+ P Q R l m (intc .l .m (inj1 x) x1) =
  intc l m (inj1 (inj1 x))
  (monadicLaw3 (PI P x) Q R l m x1)
monadicLaw3+ P Q R l m (intc .l .m (inj2 y) x1) =
  intc l m (inj1 (inj2 y))
  (monadPT+ P Q R y l m x1)
monadicLaw3+ P Q R .[] .(just (PT
```

$$(P \gg=+ (\lambda x \rightarrow Q x \gg= R)) x) (\text{terc } x) = \text{efq } x$$

--@BEGIN@monadicLawthreeInfDef

```
monadicLaw3∞ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → ((P >>=∞ Q) >>=∞ R) ⊑∞ (P >>=∞ (λ x → Q x >>= R))
```

--@END

```
monadicLaw3∞ {lu} {c0} {c1} {c2} P Q R l m q = monadicLaw3 (forcep P) Q R l m q
```

--@BEGIN@monadPTDef

```
monadPT+ : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  → (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c2))
  → (x : Tr∞ l m (PI (P >>=+ (λ x → Q x >>= R)) (inj2 y)))
  → Tr∞ l m (PI (P >>=+ Q) (inj2 y) >>=∞ R)
monadPT+ {lu} {c0} {c1} {c2} P Q R y l m tr = tr
```

--@END

--@BEGIN@monadDownSizeDef

```
monadDownSize : {lu : LUniv}(c0 c1 : Choice) (P : Process ∞ {lu} c0)(Q : ChoiceSet c0 →
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr l m (λ x → P x)
  → Tr l m (λ x → Q x)
monadDownSize c0 c1 (terminate x) Q l m tr = tr
monadDownSize c0 c1 (node x) Q l m (tnode {l = .l} {x = .m} {P = .(λ x → P x)}) x1 =
```

--@END

```
--@BEGIN@monadDownSizePlusDef
```

```
monadDownSize+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0 → Pro
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr+ l m (⟨=+⟩ P Q)
  → Tr+ l m (⟨=+⟩ P Q)
monadDownSize+ c0 c1 q P .[] .nothing (empty {P = .(⟨=+⟩ q P)}) = empty
monadDownSize+ c0 c1 q P .(Lab q x :: l) m (extc {P = .(⟨=+⟩ q P)} l .m x x1) = extc l m x (m
monadDownSize+ c0 c1 q P l m (intc {P = .(⟨=+⟩ q P)} l .m (inj1 x) x1) = intc l m (inj1 x) (m
monadDownSize+ c0 c1 q P l m (intc {P = .(⟨=+⟩ q P)} l .m (inj2 y) x1) = intc l m (inj2 y) (m
monadDownSize+ c0 c1 q P .[] .(just (PT (⟨=+⟩ q P) x)) (terc {P = .(⟨=+⟩ q P)} x) = efq x
```

```
--@END
```

```
--@BEGIN@monadDownSizePTDef
```

```
monadDownSizePT : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0 → Pro
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr∞ l m (PI (⟨=+⟩ P Q) (inj2 y))
  → Tr∞ l m (PI (⟨=+⟩ P Q) (inj2 y))
monadDownSizePT c0 c1 P Q y l m tr = tr
```

```
--@END
```

```
--@BEGIN@monadDownSizeInfDef
```

```
monadDownSize∞ : {lu : LUniv}{c0 c1 : Choice} (P : Process∞ ∞ {lu} c0)(Q : ChoiceSet c0 → Pro
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c1))
  → Tr∞ l m (⟨=∞⟩ P Q)
  → Tr∞ l m (⟨=∞⟩ P Q)
monadDownSize∞ c0 c1 P Q l m tr = tr
```

```
--@END
```

```
--@BEGIN@monadicLawInfDef
```

```

monadicLaw $\infty$  : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+  $\infty$  {lu} c0)
  (Q : ChoiceSet c0 → Process  $\infty$  {lu} c1)
  (R : ChoiceSet c1 → Process  $\infty$  {lu} c2)
  → (l : List (Label lu))
  → (x : ChoiceSet (E P))
  → (m : Maybe (ChoiceSet c2))
  → (x1 : Tr $\infty$  l m (→= $\infty$  (PE P x) (λ x2 → →= $\infty$  (Q x2) R)))
  → Tr $\infty$  l m (→= $\infty$  (→= $\infty$  (PE P x) Q) R)

--@END
monadicLaw $\infty$  P Q R l x m tr = monadicLaw3 (forcep (PE P x)) Q R l m tr

```

mutual

--@BEGIN@monadicLawRThreeDef

```

monadicLaw3R : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process  $\infty$  {lu} c0)
  (Q : ChoiceSet c0 → Process  $\infty$  {lu} c1)
  (R : ChoiceSet c1 → Process  $\infty$  {lu} c2)
  → (P  $\gg$ = (λ x → Q x  $\gg$ = R))  $\sqsubseteq$  ((P  $\gg$ = Q)  $\gg$ = R)

```

--@END

```

monadicLaw3R {lu} {c0} {c1} {c2} (terminate x) Q R l m x1 = x1
monadicLaw3R {lu} {c0} {c1} {c2} (node x) Q R l m (tnode x1) = tnode (monadicLaw3R+ x1 Q R l m)

```

--@BEGIN@monadicLawRPlusThreeDef

```

monadicLaw3R+ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process+  $\infty$  {lu} c0)
  (Q : ChoiceSet c0 → Process  $\infty$  {lu} c1)
  (R : ChoiceSet c1 → Process  $\infty$  {lu} c2)
  → (P  $\gg$ =+ (λ x → Q x  $\gg$ = R))  $\sqsubseteq$ + ((P  $\gg$ =+ Q)  $\gg$ =+ R)

```

--@END

```

monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .[] .nothing (empty {P
  = .(→= $\infty$ + { $\infty$ } {lu} {c1} {c2} (→= $\infty$ + { $\infty$ } {lu} {c0} {c1} P Q) R)) = empty

```



```

monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .(Lab P x :: l) m (extc {P
  = .(_>>=+_ {∞} {lu} {c1} {c2} (_>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m x x1)
    = extc l m x (monadicLaw3R∞ (PE P x) Q R l m x1)
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
  = .(_>>=+_ {∞} {lu} {c1} {c2} (_>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj1 (inj1 x)) x1)
    = intc l m (inj1 x) (monadicLaw3R∞ (PI P x) Q R l m x1)
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
  = .(_>>=+_ {∞} {lu} {c1} {c2} (_>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj1 (inj2 y)) x1)
    = intc l m (inj2 y) (monadPT+' P Q R y l m x1)
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R l m (intc {P
  = .(_>>=+_ {∞} {lu} {c1} {c2} (_>>=+_ {∞} {lu} {c0} {c1} P Q) R)} l .m (inj2 ()) x1)
monadicLaw3R+ {lu} {c0} {c1} {c2} P Q R .[] .(just (PT (_>>=+_ (P >>=+ Q) R) x))
  (terc {P = .(_>>=+_ (P >>=+ Q) R)} x)
  = lemTrTerminateBind''' c0 c1 c2 P Q R x

```

```
--@BEGIN@monadicLawTreeInfDef
```

```

monadicLaw3R∞ : {lu : LUniv}{c0 c1 c2 : Choice} (P : Process∞ ∞ {lu} c0)
  (Q : ChoiceSet c0 → Process ∞ {lu} c1)
  (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (P >>=∞ (λ x → Q x >>= R)) ⊑∞ ((P >>=∞ Q) >>=∞ R)

```

```
--@END
```

```
monadicLaw3R∞ {lu} {c0} {c1} {c2} P Q R l m q = monadicLaw3R (forcep P) Q R l m q
```

```
--@BEGIN@monadPTplusDef
```

```

monadPT+' : {lu : LUniv}{c0 c1 c2 : Choice}(P : Process+ ∞ {lu} c0)(Q : ChoiceSet c0 → Process
  → (R : ChoiceSet c1 → Process ∞ {lu} c2)
  → (y : ChoiceSet (T P))
  → (l : List (Label lu))
  → (m : Maybe (ChoiceSet c2))
  → (x : Tr∞ l m (_>>=∞_ (PI (_>>=+_ P Q) (inj2 y)) R))
  → Tr∞ l m (PI (_>>=+_ P (λ x → >>=_ (Q x) R)) (inj2 y))
monadPT+' {lu} {c0} {c1} {c2} P Q R y l m tr = tr

```

```
--@END
```



```

--@BEGIN@monadicLawEqThreeDef

≡monadicLaw3 : {lu : LUniv}{c0 c1 c2 : Choice}
               (P : Process ∞ {lu} c0)
               (Q : ChoiceSet c0 → Process ∞ {lu} c1)
               (R : ChoiceSet c1 → Process ∞ {lu} c2)
               → ((P >>= Q)>>= R) ≡ (P >>= (λ x → Q x >>= R))

--@END
≡monadicLaw3 {lu} {c0} {c1} {c2} P Q R = (monadicLaw3 P Q R) ,
                                              (monadicLaw3 R P Q R)

```

A.82 proofRefLaw.agda

```

--@PREFIX@mainproofRefLaw

module proofRefLaw where

open import process
open import Size
open import choiceSetU
open import Data.Product
open import RefWithoutSize
open import labelUniv
open import traceEquivalence

--@BEGIN@refProofTheo

refl⊆ : {lu : LUniv}{c : Choice} (P : Process ∞ {lu} c) → P ⊆ P
trans⊆ : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
        (Q : Process ∞ {lu} c)
        (R : Process ∞ {lu} c) → P ⊆ Q → Q ⊆ R → P ⊆ R
antiSym⊆ : {lu : LUniv}{c0 : Choice} → (P Q : Process ∞ {lu} c0)
        → P ⊆ Q → Q ⊆ P → P ≡ Q

```

```
--@END
```

```
--@BEGIN@refProof
```

```
ref⊆      P l m x      = x
trans⊆    P Q R PQ QR l m tr = PQ l m (QR l m tr)
antiSym⊆  P Q PQ QP      = PQ , QP
```

```
--@END
```

```
antiSym⊆' : {lu : LUniv}{c0 : Choice} → (P Q : Process ∞ {lu} c0) → Set
antiSym⊆' P Q = P ⊆ Q × Q ⊆ P
```

A.83 proofRefLawFdiCorrected.agda

```
--@PREFIX@mainproofRefLawFdiCorrected
```

```
module proofRefLawFdiCorrected where
```

```
open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import labelUniv
open import fdiRefusal
open import RefWithoutSize
open import traceEquivalence
open import Data.Product hiding (_×_)
open import primitiveProcess
open import auxData
open import proofRefLaw
```

```
_≡fdi1_ : {lu : LUniv}{c0 : Choice} → (P Q : Process ∞ {lu} c0) → Set
P ≡fdi1 Q = (P ⊆fdi1 Q) × (Q ⊆fdi1 P)
```

```
_≡fdi2ros_ : {lu : LUniv}{c0 : Choice} → (P Q : Process ∞ {lu} c0) → Set
P ≡fdi2ros Q = (P ⊆fdi2ros Q) × (Q ⊆fdi2ros P)
```

--@BEGIN@proofRef1TransFdiOne

$\text{refl}\sqsubseteq_{\text{fdi}_1} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c) \rightarrow P \sqsubseteq_{\text{fdi}_1} P$
 $\text{refl}\sqsubseteq_{\text{fdi}_1} P \text{ l divp} = \text{divp}$

 $\text{antiSym}\sqsubseteq_{\text{fdi}_1} : \{lu : \text{LUniv}\}\{c_0 : \text{Choice}\}$
 $\rightarrow (P Q : \text{Process} \infty \{lu\} c_0) \rightarrow P \sqsubseteq_{\text{fdi}} Q$
 $\rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q$
 $\text{antiSym}\sqsubseteq_{\text{fdi}_1} P Q PQ QP = PQ \text{ ,, } QP$

 $\text{trans}\sqsubseteq_{\text{fdi}_1} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c)$
 $(Q : \text{Process} \infty \{lu\} c)$
 $(R : \text{Process} \infty \{lu\} c)$
 $\rightarrow P \sqsubseteq_{\text{fdi}_1} Q \rightarrow Q \sqsubseteq_{\text{fdi}_1} R \rightarrow P \sqsubseteq_{\text{fdi}_1} R$
 $\text{trans}\sqsubseteq_{\text{fdi}_1} P Q R PQ QR \text{ l divp} = PQ \text{ l } (QR \text{ l divp})$

 --@END

$\text{refl}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c) \rightarrow P \sqsubseteq_{\text{fdi}_2\text{ros}} P$
 $\text{refl}\sqsubseteq_{\text{fdi}_2} P \text{ l cond fp} = \text{fp}$

 $\text{antiSym}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\}\{c_0 : \text{Choice}\}$
 $\rightarrow (P Q : \text{Process} \infty \{lu\} c_0)$
 $\rightarrow P \sqsubseteq_{\text{fdi}} Q \rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q$
 $\text{antiSym}\sqsubseteq_{\text{fdi}_2} P Q PQ QP = PQ \text{ ,, } QP$

 $\text{trans}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\}\{c : \text{Choice}\}$
 $(P : \text{Process} \infty \{lu\} c)$
 $(Q : \text{Process} \infty \{lu\} c)$
 $(R : \text{Process} \infty \{lu\} c)$
 $\rightarrow P \sqsubseteq_{\text{fdi}_2\text{ros}} Q \rightarrow Q \sqsubseteq_{\text{fdi}_2\text{ros}} R \rightarrow P \sqsubseteq_{\text{fdi}_2\text{ros}} R$
 $\text{trans}\sqsubseteq_{\text{fdi}_2} P Q R PQ QR \text{ l cond fp} = PQ \text{ l cond } (QR \text{ l cond fp})$

```

refl $\sqsubseteq$ fdi3 : {lu : LUniv}{c : Choice}
  (P : Process  $\infty$  {lu} c)  $\rightarrow$  P  $\sqsubseteq$ fdi3 P
refl $\sqsubseteq$ fdi3 P l fp = fp

antiSym $\sqsubseteq$ fdi3 : {lu : LUniv}{c0 : Choice}
   $\rightarrow$  (P Q : Process  $\infty$  {lu} c0)
   $\rightarrow$  P  $\sqsubseteq$ fdi Q  $\rightarrow$  Q  $\sqsubseteq$ fdi P  $\rightarrow$  P  $\equiv$ fdi Q
antiSym $\sqsubseteq$ fdi3 P Q PQ QP = PQ „ QP

trans $\sqsubseteq$ fdi3 : {lu : LUniv}{c : Choice}
  (P : Process  $\infty$  {lu} c)
  (Q : Process  $\infty$  {lu} c)
  (R : Process  $\infty$  {lu} c)
   $\rightarrow$  P  $\sqsubseteq$ fdi3 Q  $\rightarrow$  Q  $\sqsubseteq$ fdi3 R  $\rightarrow$  P  $\sqsubseteq$ fdi3 R
trans $\sqsubseteq$ fdi3 P Q R PQ QR l fp = PQ l (QR l fp)

```

```
--@BEGIN@proofReflTransFdi
```

```

refl $\sqsubseteq$ fdi : {lu : LUniv}{c : Choice}
  (P : Process  $\infty$  {lu} c)  $\rightarrow$  P  $\sqsubseteq$ fdi P
refl $\sqsubseteq$ fdi P = ((refl $\sqsubseteq$  P „ refl $\sqsubseteq$ fdi1 P) „ refl $\sqsubseteq$ fdi2 P) „ refl $\sqsubseteq$ fdi3 P

antiSym $\sqsubseteq$ fdi : {lu : LUniv}{c0 : Choice}
   $\rightarrow$  (P Q : Process  $\infty$  {lu} c0)  $\rightarrow$  P  $\sqsubseteq$ fdi Q
   $\rightarrow$  Q  $\sqsubseteq$ fdi P  $\rightarrow$  P  $\equiv$ fdi Q
antiSym $\sqsubseteq$ fdi P Q PQ QP = PQ „ QP

trans $\sqsubseteq$ fdi : {lu : LUniv}{c : Choice}
  (P : Process  $\infty$  {lu} c)
  (Q : Process  $\infty$  {lu} c)
  (R : Process  $\infty$  {lu} c)
   $\rightarrow$  P  $\sqsubseteq$ fdi Q  $\rightarrow$  Q  $\sqsubseteq$ fdi R  $\rightarrow$  P  $\sqsubseteq$ fdi R
trans $\sqsubseteq$ fdi P Q R (((PQ „ PQfdi1) „ PQfdi2) „ PQfdi3)
  (((QR „ QRfdi1) „ QRfdi2) „ QRfdi3)
= ((( trans $\sqsubseteq$  P Q R PQ QR
  „ trans $\sqsubseteq$ fdi1 P Q R PQfdi1 QRfdi1 )
  „ trans $\sqsubseteq$ fdi2 P Q R PQfdi2 QRfdi2 )

```

```
„ trans $\sqsubseteq$ fdi3 P Q R PQfdi3 QRfdi3 )
```

```
--@END
```

A.84 proofRefLawFdiModified.agda

```
--@PREFIX@mainproofRefLawFdiModified
```

```
module proofRefLawFdiModified where
```

```
open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import labelUniv
open import fdiPart2
open import RefWithoutSize
open import traceEquivalence
open import Data.Product hiding (_ $\times$ _ )
open import primitiveProcess
open import auxData
open import proofRefLaw
```

```
 $\_ \equiv \text{fdi}_1 \_ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow \text{Set}$   

 $P \equiv \text{fdi}_1 \ Q = (P \sqsubseteq \text{fdi}_1 \ Q) \times (Q \sqsubseteq \text{fdi}_1 \ P)$ 
```

```
 $\_ \equiv \text{fdi}_2 \_ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow \text{Set}$   

 $P \equiv \text{fdi}_2 \ Q = (P \sqsubseteq \text{fdi}_2 \ Q) \times (Q \sqsubseteq \text{fdi}_2 \ P)$ 
```

```
--@BEGIN@proofRefLTransFdiOne
```

```
refl $\sqsubseteq$ fdi1 : {lu : LUniv} {c : Choice}  

             (P : Process  $\infty$  {lu} c)  $\rightarrow$  P  $\sqsubseteq$ fdi1 P  

refl $\sqsubseteq$ fdi1 P l divp = divp
```

```
antiSym $\sqsubseteq$ fdi1 : {lu : LUniv} {c0 : Choice}
```

$$\begin{aligned} &\rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow P \sqsubseteq_{\text{fdi}} Q \\ &\rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q \\ \text{antiSym}\sqsubseteq_{\text{fdi}_1} P \ Q \ P Q \ QP &= P Q \text{ ,, } QP \end{aligned}$$

$$\begin{aligned} \text{trans}\sqsubseteq_{\text{fdi}_1} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \\ & (Q : \text{Process} \infty \{lu\} \ c) \\ & (R : \text{Process} \infty \{lu\} \ c) \\ & \rightarrow P \sqsubseteq_{\text{fdi}_1} Q \rightarrow Q \sqsubseteq_{\text{fdi}_1} R \rightarrow P \sqsubseteq_{\text{fdi}_1} R \\ \text{trans}\sqsubseteq_{\text{fdi}_1} P \ Q \ R \ P Q \ Q R \ l \ \text{divp} &= P Q \ l \ (Q R \ l \ \text{divp}) \end{aligned}$$

--@END

$$\begin{aligned} \text{refl}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \rightarrow P \sqsubseteq_{\text{fdi}_2} P \\ \text{refl}\sqsubseteq_{\text{fdi}_2} P \ l \ \text{cond} \ \text{fp} &= \text{fp} \end{aligned}$$

$$\begin{aligned} \text{antiSym}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \\ & \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \\ & \rightarrow P \sqsubseteq_{\text{fdi}} Q \rightarrow Q \sqsubseteq_{\text{fdi}} P \rightarrow P \equiv_{\text{fdi}} Q \\ \text{antiSym}\sqsubseteq_{\text{fdi}_2} P \ Q \ P Q \ QP &= P Q \text{ ,, } QP \end{aligned}$$

$$\begin{aligned} \text{trans}\sqsubseteq_{\text{fdi}_2} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \\ & (Q : \text{Process} \infty \{lu\} \ c) \\ & (R : \text{Process} \infty \{lu\} \ c) \\ & \rightarrow P \sqsubseteq_{\text{fdi}_2} Q \rightarrow Q \sqsubseteq_{\text{fdi}_2} R \rightarrow P \sqsubseteq_{\text{fdi}_2} R \\ \text{trans}\sqsubseteq_{\text{fdi}_2} P \ Q \ R \ P Q \ Q R \ l \ \text{cond} \ \text{fp} &= P Q \ l \ \text{cond} \ (Q R \ l \ \text{cond} \ \text{fp}) \end{aligned}$$

--@BEGIN@proofRef1TransFdi

$$\begin{aligned} \text{refl}\sqsubseteq_{\text{fdi}} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} \\ & (P : \text{Process} \infty \{lu\} \ c) \rightarrow P \sqsubseteq_{\text{fdi}} P \\ \text{refl}\sqsubseteq_{\text{fdi}} P &= (\text{refl}\sqsubseteq P \text{ ,, } \text{refl}\sqsubseteq_{\text{fdi}_1} P) \text{ ,, } \text{refl}\sqsubseteq_{\text{fdi}_2} P \end{aligned}$$

$$\begin{aligned} \text{antiSym}\sqsubseteq_{\text{fdi}} : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \\ & \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow P \sqsubseteq_{\text{fdi}} Q \end{aligned}$$

```

      → Q ⊑fdi P → P ≡fdi Q
antiSym⊑fdi P Q PQ QP = PQ „ QP

trans⊑fdi : {lu : LUniv}{c : Choice}
  (P : Process ∞ {lu} c)
  (Q : Process ∞ {lu} c)
  (R : Process ∞ {lu} c)
  → P ⊑fdi Q → Q ⊑fdi R → P ⊑fdi R
trans⊑fdi P Q R ((PQ „ PQfdi1) „ PQfdi2)
  ((QR „ QRfdi1) „ QRfdi2)
= ( trans⊑ P Q R PQ QR
  „ trans⊑fdi1 P Q R PQfdi1 QRfdi1 )
  „ trans⊑fdi2 P Q R PQfdi2 QRfdi2

--@END

```

A.85 proofRenamingSkip.agda

```
--@PREFIX@mainproofRenamingSkip
```

```
module proofRenamingSkip where
```

```

open import process
open import Size
open import choiceSetU
open import labelUniv
open import TraceWithoutSize
open import RefWithoutSize
open import primitiveProcess
open import renamingOperator
open import traceEquivalence
open import Data.Product

```

```
--@BEGIN@unitRenameLaw
```

```

unitRenameLaw : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Label lu)
  → (P : Process i {lu} c0)
  → Rename A (terminate a) ⊆ (terminate a)
unitRenameLaw a A P l m x = x

```

```

unitRenameLawr : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Label lu)
  → (P : Process i {lu} c0)
  → (terminate a) ⊆ Rename A (terminate a)
unitRenameLawr a A P l m x = x

```

```
--@END
```

```
--@BEGIN@unitRenameLawTheo
```

```

≡unitRename : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Label lu)
  → (P : Process i {lu} c0)
  → Rename A (terminate a) ≡ (terminate a)

```

```
--@END
```

```
--@BEGIN@unitRenameLawTheoProof
```

```

≡unitRename a A P = (unitRenameLaw a A P) , (unitRenameLawr a A P)

```

```
--@END
```

A.86 proofSymExt.agda

```
--@PREFIX@mainproofSymExt
```

```

module proofSymExt where
open import process
open import Size

```

```

open import choiceSetU
open import Data.Maybe
open import Data.Fin
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import dataAuxFunction
open import externalChoice
open import addTick
open import internalChoice
open import lemFmap
open import traceEquivalence
open import Data.Product
open import labelUniv

```

```
mutual
```

```
--@BEGIN@SymExCPDef
```

```

S□+ : {lu : LUniv}{c₀ c₁ : Choice} (P : Process+ ∞ {lu} c₀)
      (Q : Process+ ∞ {lu} c₁)
      → (P □++ Q) ⊑+ (fmap+ swap⊕ (Q □++ P))
S□+ P Q .[] .nothing empty = empty
S□+ P Q .(Lab Q x :: l) m (extc l .m (inj₁ x) x₁) = extc l m (inj₂ x)
      (lemFmap∞ inj₁ swap⊕ (PE Q x) l m x₁)
S□+ P Q .(Lab P y :: l) m (extc l .m (inj₂ y) x₁) = extc l m (inj₁ y)
      (lemFmap∞ inj₂ swap⊕ (PE P y) l m x₁)
S□+ P Q l m (intc .l .m (inj₁ x) x₁) = intc l m (inj₂ x)
      (S□+∞ P (PI Q x) l m x₁)
S□+ P Q l m (intc .l .m (inj₂ y) x₁) = intc l m (inj₁ y)
      (S□+∞ (PI P y) Q l m x₁)
S□+ P Q .[] .(just (inj₂ (PT Q x))) (terc (inj₁ x)) = terc (inj₂ x)
S□+ P Q .[] .(just (inj₁ (PT P y))) (terc (inj₂ y)) = terc (inj₁ y)

```

```
--@END
```

```

S□+∞ : {lu : LUniv}{c₀ c₁ : Choice}
      → (P : Process+ ∞ {lu} c₀)

```

```

    → (Q : Process∞ ∞ c₁)
    → Ref∞ (P □+∞+ Q) (fmap∞ swap⊕ (Q □∞++ P))
S□+∞ P Q l m (tnode tr) = tnode (S□+p P (forcep Q) l m tr)

S□+p : {lu : LUniv}{c₀ c₁ : Choice}
    → (P : Process+ ∞ {lu} c₀)
    → (Q : Process ∞ c₁)
    → Ref+ (P □+p+ Q) (fmap+ swap⊕ (Q □p++ P))
S□+p P (terminate x) l m q = addTimeFmapLemma+ inj₂ swap⊕ P (inj₁ x) l m q
S□+p P (node Q) l m q = S□+ P Q l m q

```

```

S□∞+ : {lu : LUniv}{c₀ c₁ : Choice}
    → (P : Process∞ ∞ c₀)
    → (Q : Process+ ∞ {lu} c₁)
    → Ref∞ (P □∞++ Q) (fmap∞ swap⊕ (Q □+∞+ P))
S□∞+ P Q l m (tnode tr) = tnode (S□p+ (forcep P) Q l m tr)

```

```

S□p+ : {lu : LUniv}{c₀ c₁ : Choice}
    → (P : Process ∞ c₀)
    → (Q : Process+ ∞ {lu} c₁)
    → Ref+ (P □p++ Q) (fmap+ swap⊕ (Q □+p+ P))
S□p+ (terminate x) P l m q = addTimeFmapLemma+ inj₁ swap⊕ P (inj₂ x) l m q
S□p+ (node Q) P l m q = S□+ Q P l m q

```

mutual

```

S□+R : {lu : LUniv}{c₀ c₁ : Choice} (P : Process+ ∞ {lu} c₀) (Q : Process+ ∞ {lu} c₁)
    → (fmap+ swap⊕ (Q □++ P)) □+ (P □++ Q)
S□+R P Q .[] .nothing empty = empty
S□+R P Q .(Lab P x :: l) m (extc l m (inj₁ x) x₁) = extc l m (inj₂ x) (lemFmap∞R inj₂ swap⊕ (PE
S□+R P Q .(Lab Q y :: l) m (extc l m (inj₂ y) x₁) = extc l m (inj₁ y) (lemFmap∞R inj₁ swap⊕ (PE
S□+R P Q l m (intc .l m (inj₁ x) x₁) = intc l m (inj₂ x) (S□∞+R (PI P x) Q l m x₁)
S□+R P Q l m (intc .l m (inj₂ y) x₁) = intc l m (inj₁ y) (S□+∞R P (PI Q y) l m x₁)
S□+R P Q .[] .(just (inj₁ (PT P x))) (terc (inj₁ x)) = (terc (inj₂ x))
S□+R P Q .[] .(just (inj₂ (PT Q y))) (terc (inj₂ y)) = (terc (inj₁ y))

```

```

S□+∞R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process+ ∞ {lu} c₀)
  → (Q : Process∞ ∞ c₁)
  → Ref∞ (fmap∞ swap⊕ (Q □∞++ P)) (P □+∞+ Q)
S□+∞R P Q l m (tnode tr) = tnode (S□+pR P (forcep Q) l m tr)

S□+pR : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process+ ∞ {lu} c₀)
  → (Q : Process ∞ c₁)
  → Ref+ (fmap+ swap⊕ (Q □p++ P)) (P □+p+ Q)
S□+pR P (terminate x) l m q = addTimeFmapLemma+R inj₂ swap⊕ P (inj₁ x) l m q
S□+pR P (node Q) l m q = S□+R P Q l m q

S□∞+R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process∞ ∞ c₀)
  → (Q : Process+ ∞ {lu} c₁)
  → Ref∞ (fmap∞ swap⊕ (Q □+∞+ P)) (P □∞++ Q)
S□∞+R P Q l m (tnode tr) = tnode (S□p+R (forcep P) Q l m tr)

S□p+R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process ∞ c₀)
  → (Q : Process+ ∞ {lu} c₁)
  → Ref+ (fmap+ swap⊕ (Q □+p+ P)) (P □p++ Q)
S□p+R (terminate x) P l m q = addTimeFmapLemma+R inj₁ swap⊕ P (inj₂ x) l m q
S□p+R (node Q) P l m q = S□+R Q P l m q

```

```
--@BEGIN@SymExCPEqDef
```

```

≡□+ : {lu : LUniv}{c₀ c₁ : Choice} (P : Process+ ∞ {lu} c₀)
  (Q : Process+ ∞ {lu} c₁)
  → (P □++ Q) ≡+ (fmap+ swap⊕ (Q □++ P))
≡□+ P Q = S□+ P Q , S□+R P Q

```

```
--@END
```

A.87 proofSymInterleaving.agda

```
--@PREFIX@mainproofSymInterleaving

module proofSymInterleaving where

open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import lemFmap
open import auxData
open import Data.Product
open import Interleave
open import traceEquivalence
open import Data.Product
open import labelUniv

mutual
--@BEGIN@interleavProofCDef

S|||+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
  → (P |||++ Q) ⊆+ (fmap+ swap× (Q |||++ P))
S|||+ P Q .[] .nothing empty = empty
S|||+ P Q .(Lab Q x :: l) m (extc l .m (inj1 x) q) =
  extc l m (inj2 x)(S|||+∞ P (PE Q x) l m q)
S|||+ P Q .(Lab P x :: l) m (extc l .m (inj2 x) q) =
  extc l m (inj1 x)(S|||+∞+ (PE P x) Q l m q)
S|||+ P Q l m (intc l .m (inj1 x) q) =
  intc l m (inj2 x)(S|||+∞ P (PI Q x) l m q)
```

```

S|||+ P Q l m (intc .l .m (inj2 x) q) =
  intc l m (inj1 x)(S|||∞+ (PI P x) Q l m q)
S|||+ P Q .[] .(just (PT P x ,, PT Q y)) (terc (y ,, x)) =
  terc (x ,, y)

```

```
--@END
```

```

S||| : {lu : LUniv}{c0 c1 : Choice} → (P : Process ∞ {lu} c0)
      → (Q : Process ∞ {lu} c1)
      → (P ||| Q) ⊆ (fmap swap× (Q ||| P))
S||| (terminate x) (terminate x') l m q = q
S||| (terminate x) (node P) l m (tnode q) =
  tnode (lemFmap+ (λ a → a ,, x) swap× P l m q)
S||| (node P) (terminate x) l m (tnode q) =
  tnode (lemFmap+ (λ a → a ,, x) swap× P l m q)
S||| (node P) (node Q) l m (tnode q) = tnode (S|||+ P Q l m q)

S|||+p : {lu : LUniv}{c0 c1 : Choice}
        → (P : Process+ ∞ {lu} c0)
        → (Q : Process ∞ c1)
        → Ref+ (P |||+p Q) (fmap+ swap× (Q |||+p P))
S|||+p P (terminate x) l m q = lemFmap+ (λ a → a ,, x) swap× P l m q
S|||+p P (node Q) l m q = S|||+ P Q l m q

```

```

S|||p+ : {lu : LUniv}{c0 c1 : Choice}
        → (P : Process ∞ {lu} c0)
        → (Q : Process+ ∞ {lu} c1)
        → Ref+ (P |||p+ Q) (fmap+ swap× (Q |||+p P))
S|||p+ (terminate x) Q l m q = lemFmap+ (λ a → a ,, x) swap× Q l m q
S|||p+ (node P) Q l m q = S|||+ P Q l m q

```

```

S|||+∞ : {lu : LUniv}{c0 c1 : Choice}
        → (P : Process+ ∞ {lu} c0)
        → (Q : Process∞ ∞ c1)
        → Ref∞ (P |||+∞ Q) (fmap∞ swap× (Q |||∞+ P))
S|||+∞ P Q l m (tnode tr) = tnode (S|||+p P (forcep Q) l m tr)

```

```

S|||∞+ : {lu : LUniv}{c0 c1 : Choice}
        → (P : Process∞ ∞ {lu} c0)

```

```

→ (Q : Process+ ∞ c₁)
→ Ref∞ (P |||∞+ Q) (fmap∞ swap× (Q |||+∞ P))
S|||∞+ P Q l m (tnode tr) = tnode (S|||p+ (forcep P) Q l m tr)

```

mutual

```

S|||+R : {lu : LUniv}{c₀ c₁ : Choice} → (P : Process+ ∞ {lu} c₀) → (Q : Process+ ∞ {lu} c₁)
→ (fmap+ swap× (Q |||++ P)) ⊑+ (P |||++ Q)
S|||+R P Q .[] .nothing empty = empty
S|||+R P Q .(Lab P x :: l) m (extc l .m (inj₁ x) x₁) = extc l m (inj₂ x) (S|||∞+R (PE P x) Q l m x₁)
S|||+R P Q .(Lab Q y :: l) m (extc l .m (inj₂ y) x₁) = extc l m (inj₁ y) (S|||+∞R P (PE Q y) l m x₁)
S|||+R P Q l m (intc l .m (inj₁ x) x₁) = intc l m (inj₂ x) (S|||∞+R (PI P x) Q l m x₁)
S|||+R P Q l m (intc l .m (inj₂ y) x₁) = intc l m (inj₁ y) (S|||+∞R P (PI Q y) l m x₁)
S|||+R P Q .[] .(just (PT P x ,, PT Q x₁)) (terc (x ,, x₁)) = terc (x₁ ,, x)

```

```

S|||R : {lu : LUniv}{c₀ c₁ : Choice} → (P : Process ∞ {lu} c₀) → (Q : Process ∞ {lu} c₁)
→ (fmap swap× (Q ||| P)) ⊑ (P ||| Q)
S|||R (terminate x) (terminate x') l m q = q
S|||R (terminate x) (node P) l m (tnode q) = tnode (lemFmap+R (λ a → a ,, x) swap× P l m q)
S|||R (node P) (terminate x) l m (tnode q) = tnode (lemFmap+R (λ a → a ,, x) swap× P l m q)
S|||R (node P) (node Q) l m (tnode q) = tnode (S|||+R P Q l m q)

```

```

S|||+pR : {lu : LUniv}{c₀ c₁ : Choice}
→ (P : Process+ ∞ {lu} c₀)
→ (Q : Process ∞ c₁)
→ Ref+ (fmap+ swap× (Q |||p+ P)) (P |||p+ Q)
S|||+pR P (terminate x) l m q = lemFmap+R (λ a → a ,, x) swap× P l m q
S|||+pR P (node Q) l m q = S|||+R P Q l m q

```

```

S|||p+R : {lu : LUniv}{c₀ c₁ : Choice}
→ (P : Process ∞ {lu} c₀)
→ (Q : Process+ ∞ {lu} c₁)
→ Ref+ (fmap+ swap× (Q |||p+ P)) (P |||p+ Q)
S|||p+R (terminate x) Q l m q = lemFmap+R (λ a → a ,, x) swap× Q l m q
S|||p+R (node P) Q l m q = S|||+R P Q l m q

```

```

S|||+∞R : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process+ ∞ {lu} c0)
  → (Q : Process∞ ∞ c1)
  → Ref∞ (fmap∞ swap× (Q |||∞+ P)) (P |||+∞ Q)
S|||+∞R P Q l m (tnode tr) = tnode (S|||+pR P (forcep Q) l m tr)

```

```

S|||∞+R : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process∞ ∞ {lu} c0)
  → (Q : Process+ ∞ c1)
  → Ref∞ (fmap∞ swap× (Q |||+∞ P)) (P |||∞+ Q)
S|||∞+R P Q l m (tnode tr) = tnode (S|||p+R (forcep P) Q l m tr)

```

```
--@BEGIN@interleavProofCEqDef
```

```

≡S|||+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
  (Q : Process+ ∞ {lu} c1)
  → (P |||++ Q) ≡+ (fmap+ swap× (Q |||++ P))
≡S|||+ P Q = (S|||+ P Q) , (S|||+R P Q)

```

```
--@END
```

A.88 proofSymInterleavingTheoOnly.agda

```
--@PREFIX@mainproofSymInterleavingTheoOnly
```

```
module proofSymInterleavingTheoOnly where
```

```

open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize

```

```

open import RefWithoutSize
open import lemFmap
open import auxData
open import Data.Product
open import Interleave
open import traceEquivalence
open import Data.Product
open import labelUniv

```

```
mutual
```

```
--@BEGIN@interleavProofCDef
```

```

S|||+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (Q : Process+ ∞ {lu} c1)
  → (P |||++ Q) ⊆+ (fmap+ swap× (Q |||++ P))
S|||+ P Q .[] .nothing empty = empty
S|||+ P Q .(Lab Q x :: l) m (extc l .m (inj1 x) q) =
  extc l m (inj2 x)(S|||+∞ P (PE Q x) l m q)
S|||+ P Q .(Lab P x :: l) m (extc l .m (inj2 x) q) =
  extc l m (inj1 x)(S|||+∞+ (PE P x) Q l m q)
S|||+ P Q l m (intc l .m (inj1 x) q) =
  intc l m (inj2 x)(S|||+∞ P (PI Q x) l m q)
S|||+ P Q l m (intc l .m (inj2 x) q) =
  intc l m (inj1 x)(S|||+∞+ (PI P x) Q l m q)
S|||+ P Q .[] .(just (PT P x ,, PT Q y)) (terc (y ,, x)) =
  terc (x ,, y)

```

```
--@END
```

```

S||| : {lu : LUniv}{c0 c1 : Choice} → (P : Process ∞ {lu} c0)
      → (Q : Process ∞ {lu} c1)
      → (P ||| Q) ⊆ (fmap swap× (Q ||| P))
S||| (terminate x) (terminate x') l m q = q
S||| (terminate x) (node P) l m (tnode q) =
  tnode (lemFmap+ (λ a → a ,, x) swap× P l m q)
S||| (node P) (terminate x) l m (tnode q) =
  tnode (lemFmap+ (λ a → a ,, x) swap× P l m q)
S||| (node P) (node Q) l m (tnode q) = tnode (S|||+ P Q l m q)

```

$S|||+p : \{lu : LUniv\}\{c_0\ c_1 : Choice\}$
 $\rightarrow (P : Process+ \infty \{lu\}\ c_0)$
 $\rightarrow (Q : Process \infty c_1)$
 $\rightarrow Ref+ (P |||+p\ Q)(fmap+ swap \times (Q |||+p\ P))$
 $S|||+p\ P\ (terminate\ x)\ l\ m\ q = lemFmap+ (_, -\ x)\ swap \times P\ l\ m\ q$
 $S|||+p\ P\ (node\ Q)\ l\ m\ q = S|||+ P\ Q\ l\ m\ q$

$S|||p+ : \{lu : LUniv\}\{c_0\ c_1 : Choice\}$
 $\rightarrow (P : Process \infty \{lu\}\ c_0)$
 $\rightarrow (Q : Process+ \infty \{lu\}\ c_1)$
 $\rightarrow Ref+ (P |||p+\ Q)\ (fmap+ swap \times (Q |||+p\ P))$
 $S|||p+\ (terminate\ x)\ Q\ l\ m\ q = lemFmap+ (\lambda\ a \rightarrow a\ _, x)\ swap \times Q\ l\ m\ q$
 $S|||p+\ (node\ P)\ Q\ l\ m\ q = S|||+ P\ Q\ l\ m\ q$

$S|||+\infty : \{lu : LUniv\}\{c_0\ c_1 : Choice\}$
 $\rightarrow (P : Process+ \infty \{lu\}\ c_0)$
 $\rightarrow (Q : Process \infty c_1)$
 $\rightarrow Ref\infty (P |||+\infty\ Q)\ (fmap\infty swap \times (Q |||\infty+ P))$
 $S|||+\infty\ P\ Q\ l\ m\ (tnode\ tr) = tnode\ (S|||+p\ P\ (forcep\ Q)\ l\ m\ tr)$

$S|||\infty+ : \{lu : LUniv\}\{c_0\ c_1 : Choice\}$
 $\rightarrow (P : Process\infty \infty \{lu\}\ c_0)$
 $\rightarrow (Q : Process+ \infty c_1)$
 $\rightarrow Ref\infty (P |||\infty+ Q)\ (fmap\infty swap \times (Q |||+\infty P))$
 $S|||\infty+ P\ Q\ l\ m\ (tnode\ tr) = tnode\ (S|||p+\ (forcep\ P)\ Q\ l\ m\ tr)$

mutual

$S|||+R : \{lu : LUniv\}\{c_0\ c_1 : Choice\} \rightarrow (P : Process+ \infty \{lu\}\ c_0) \rightarrow (Q : Process+ \infty \{lu\}\ c_1)$
 $\rightarrow (fmap+ swap \times (Q |||++ P)) \sqsubseteq+ (P |||++ Q)$
 $S|||+R\ P\ Q\ .[]\ .nothing\ empty = empty$
 $S|||+R\ P\ Q\ .(Lab\ P\ x :: l)\ m\ (extc\ l\ m\ (inj_1\ x)\ x_1) = extc\ l\ m\ (inj_2\ x)\ (S|||\infty+R\ (PE\ P\ x)\ Q\ l\ m\ x_1)$
 $S|||+R\ P\ Q\ .(Lab\ Q\ y :: l)\ m\ (extc\ l\ m\ (inj_2\ y)\ x_1) = extc\ l\ m\ (inj_1\ y)\ (S|||+\infty R\ P\ (PE\ Q\ y)\ l\ m\ x_1)$
 $S|||+R\ P\ Q\ l\ m\ (intc\ l\ m\ (inj_1\ x)\ x_1) = intc\ l\ m\ (inj_2\ x)\ (S|||\infty+R\ (PI\ P\ x)\ Q\ l\ m\ x_1)$
 $S|||+R\ P\ Q\ l\ m\ (intc\ l\ m\ (inj_2\ y)\ x_1) = intc\ l\ m\ (inj_1\ y)\ (S|||+\infty R\ P\ (PI\ Q\ y)\ l\ m\ x_1)$

$S|||+R \ P \ Q \ .[] \ .(\text{just } (\text{PT } P \ x \ , \ \text{PT } Q \ x_1)) \ (\text{terc } (x \ , \ x_1)) = \text{terc } (x_1 \ , \ x)$

$S|||R : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\} \rightarrow (P : \text{Process} \ \infty \ \{lu\} \ c_0) \rightarrow (Q : \text{Process} \ \infty \ \{lu\} \ c_1)$
 $\rightarrow (\text{fmap } \text{swap} \times (Q ||| P)) \sqsubseteq (P ||| Q)$

$S|||R \ (\text{terminate } x) \ (\text{terminate } x') \ l \ m \ q = q$

$S|||R \ (\text{terminate } x) \ (\text{node } P) \ l \ m \ (\text{tnode } q) = \text{tnode } (\text{lemFmap}+R \ (\lambda \ a \rightarrow a \ , \ x) \ \text{swap} \times \ P \ l \ m \ q)$

$S|||R \ (\text{node } P) \ (\text{terminate } x) \ l \ m \ (\text{tnode } q) = \text{tnode } (\text{lemFmap}+R \ (_, \ x) \ \text{swap} \times \ P \ l \ m \ q)$

$S|||R \ (\text{node } P) \ (\text{node } Q) \ l \ m \ (\text{tnode } q) = \text{tnode } (S|||+R \ P \ Q \ l \ m \ q)$

$S|||+pR : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\}$
 $\rightarrow (P : \text{Process}+ \ \infty \ \{lu\} \ c_0)$
 $\rightarrow (Q : \text{Process} \ \infty \ c_1)$
 $\rightarrow \text{Ref}+ \ (\text{fmap}+ \ \text{swap} \times (Q |||p+ P)) \ (P |||+p \ Q)$

$S|||+pR \ P \ (\text{terminate } x) \ l \ m \ q = \text{lemFmap}+R \ (_, \ x) \ \text{swap} \times \ P \ l \ m \ q$

$S|||+pR \ P \ (\text{node } Q) \ l \ m \ q = S|||+R \ P \ Q \ l \ m \ q$

$S|||p+R : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\}$
 $\rightarrow (P : \text{Process} \ \infty \ \{lu\} \ c_0)$
 $\rightarrow (Q : \text{Process}+ \ \infty \ \{lu\} \ c_1)$
 $\rightarrow \text{Ref}+ \ (\text{fmap}+ \ \text{swap} \times (Q |||+p \ P)) \ (P |||p+ \ Q)$

$S|||p+R \ (\text{terminate } x) \ Q \ l \ m \ q = \text{lemFmap}+R \ (\lambda \ a \rightarrow a \ , \ x) \ \text{swap} \times \ Q \ l \ m \ q$

$S|||p+R \ (\text{node } P) \ Q \ l \ m \ q = S|||+R \ P \ Q \ l \ m \ q$

$S|||+\infty R : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\}$
 $\rightarrow (P : \text{Process}+ \ \infty \ \{lu\} \ c_0)$
 $\rightarrow (Q : \text{Process} \ \infty \ c_1)$
 $\rightarrow \text{Ref} \ (\text{fmap} \ \text{swap} \times (Q |||\infty+ P)) \ (P |||+\infty \ Q)$

$S|||+\infty R \ P \ Q \ l \ m \ (\text{tnode } tr) = \text{tnode } (S|||+pR \ P \ (\text{forcep } Q) \ l \ m \ tr)$

$S|||\infty+R : \{lu : \text{LUniv}\} \{c_0 \ c_1 : \text{Choice}\}$
 $\rightarrow (P : \text{Process} \ \infty \ \{lu\} \ c_0)$
 $\rightarrow (Q : \text{Process}+ \ \infty \ c_1)$
 $\rightarrow \text{Ref} \ (\text{fmap} \ \text{swap} \times (Q |||+\infty \ P)) \ (P |||\infty+ \ Q)$

$S|||\infty+R \ P \ Q \ l \ m \ (\text{tnode } tr) = \text{tnode } (S|||p+R \ (\text{forcep } P) \ Q \ l \ m \ tr)$

--@BEGIN@interleavProofCEqDef

```

≡S|||+ : {lu : LUniv}{c0 c1 : Choice}
          (P : Process+ ∞ {lu} c0)
          (Q : Process+ ∞ {lu} c1)
          → (P |||++ Q) ≡+ (fmap+ swap× (Q |||++ P))

--@END
≡S|||+ P Q = (S|||+ P Q) , (S|||+R P Q)

```

A.89 proofSymParPartone.agda

```
--@PREFIX@mainproofSymParPartone
```

```
module proofSymParPartone where
```

```

open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import lemFmap
open import auxData
open import labelUniv
open import parallelSimple
open import restrict
open import Data.Bool.Base renaming (T to T')
open import auxLemmaPar

```

```
mutual
```

```
--@BEGIN@SymParPlusDef
```

```

S|||+ : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
      (A B : Label lu → Bool)

```

$$(Q : \text{Process} + \infty \{lu\} c_1) \\ \rightarrow (P [A] || + [B] Q) \sqsubseteq + \text{fmap} + \text{swap} \times ((Q [B] || + [A] P))$$

$$\begin{aligned} S[[]] + P A B Q . [] . \text{nothing empty} &= \text{empty} \\ S[[]] + P A B Q . (\text{Lab } Q a :: l) m (\text{extc } l . m (\text{inj}_1 (\text{inj}_1 (\text{sub } a x))) x_1) &= \\ \text{extc } l m (\text{inj}_1 (\text{inj}_2 (\text{sub } a x))) (S[[]] + \infty P A B (\text{PE } Q a) l m x_1) &= \\ S[[]] + P A B Q . (\text{Lab } P a :: l) m (\text{extc } l . m (\text{inj}_1 (\text{inj}_2 (\text{sub } a x))) x_1) &= \\ \text{extc } l m (\text{inj}_1 (\text{inj}_1 (\text{sub } a x))) (S[[]] \infty + (\text{PE } P a) A B Q l m x_1) &= \\ S[[]] + \{lu\} P A B Q . (\text{Lab } Q x :: l) m (\text{extc } l . m (\text{inj}_2 (\text{sub } (x ,, x_1) x_2))) x_3 &= \end{aligned}$$

$$\begin{aligned} &\text{let} \\ lxl x_1 : T' (\text{Lab } Q x ==| \text{Lab } P x_1) &= \\ lxl x_1 = \text{lemmaBool} (\text{Lab } Q x ==| \text{Lab } P x_1) &= \\ (B (\text{Lab } Q x) \wedge A (\text{Lab } P x_1)) x_2 &= \end{aligned}$$

$$\begin{aligned} BQx : T' (B (\text{Lab } Q x)) &= \\ BQx = \text{lemmaBool} (B (\text{Lab } Q x)) (A (\text{Lab } P x_1)) &= \\ (\text{lemmaBoolR} ((\text{Lab } Q x ==| \text{Lab } P x_1))) &= \\ (B (\text{Lab } Q x) \wedge A (\text{Lab } P x_1)) x_2 &= \end{aligned}$$

$$\begin{aligned} APx_1 : T' (A (\text{Lab } P x_1)) &= \\ APx_1 = \text{lemmaBool}' ((\text{Lab } Q x ==| \text{Lab } P x_1)) &= \\ (B (\text{Lab } Q x)) (A (\text{Lab } P x_1)) x_2 &= \end{aligned}$$

$$\begin{aligned} lx_1 lx : T' (\text{Lab } P x_1 ==| \text{Lab } Q x) &= \\ lx_1 lx = \text{sym} ==| \{lu\} \{\text{Lab } Q x\} \{\text{Lab } P x_1\} lxl x_1 &= \end{aligned}$$

$$\begin{aligned} x_2' : T' ((\text{Lab } P x_1 ==| \text{Lab } Q x) &= \\ \wedge A (\text{Lab } P x_1) \wedge B (\text{Lab } Q x)) &= \\ x_2' = \text{lemmaBool}'' (\text{Lab } P x_1 ==| \text{Lab } Q x) &= \\ (A (\text{Lab } P x_1)) (B (\text{Lab } Q x)) &= \\ lxl lx APx_1 BQx &= \end{aligned}$$

$$\begin{aligned} \text{auxproof} : \text{Tr} + (\text{Lab } P x_1 :: l) m &= \\ (P [A] || + [B] Q) &= \\ \text{auxproof} = \text{extc } l m (\text{inj}_2 (\text{sub } (x_1 ,, x) x_2')) &= \\ (S[[]] \infty \infty (\text{PE } P x_1) A B (\text{PE } Q x) l m x_3) &= \end{aligned}$$

$$\text{auxproof}' : \text{Tr} + (\text{Lab } Q x :: l) m (P [A] || + [B] Q)$$

$$\begin{aligned} auxproof' = & \text{transfLu } \{lu\} (\lambda l' \rightarrow \text{Tr+ } (l' :: l) \\ & m (P [A] || + [B] Q)) \{ \text{Lab } P x_1 \} \\ & \{ \text{Lab } Q x \} \text{ } lx_1 lx \text{ } auxproof \end{aligned}$$

$$\text{in } auxproof'$$

$$\begin{aligned} S[[]]+ P A B Q l m (\text{intc } .l .m (\text{inj}_1 x) x_1) = & \\ & \text{intc } l m (\text{inj}_2 x) (S[[]]+\infty P A B (\text{PI } Q x) l m x_1) \\ S[[]]+ P A B Q l m (\text{intc } .l .m (\text{inj}_2 y) x_1) = & \\ & \text{intc } l m (\text{inj}_1 y) (S[[]]+\infty (\text{PI } P y) A B Q l m x_1) \\ S[[]]+ P A B Q .[] .(\text{just } (\text{PT } P x_1 ,, \text{PT } Q x)) (\text{terc } (x ,, x_1)) = & \text{terc } (x_1 ,, x) \end{aligned}$$

--@END

$$\begin{aligned} S[[]]+\infty : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} & \\ \rightarrow (P : \text{Process+ } \infty \{lu\} c_0) & \\ \rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) & \\ \rightarrow (Q : \text{Process}\infty \infty c_1) & \\ \rightarrow \text{Ref}\infty (P [A] || + \infty [B] Q) (\text{fmap}\infty \text{swap} \times ((Q [B] || \infty + [A] P))) & \\ S[[]]+\infty P A B Q l m (\text{tnode } tr) = \text{tnode } (S[[]]+p P A B (\text{forcep } Q) l m tr) & \end{aligned}$$

$$\begin{aligned} S[[]]+\infty : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} & \\ \rightarrow (P : \text{Process}\infty \infty c_0) & \\ \rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) & \\ \rightarrow (Q : \text{Process+ } \infty \{lu\} c_1) & \\ \rightarrow \text{Ref}\infty (P [A] || \infty + [B] Q) (\text{fmap}\infty \text{swap} \times ((Q [B] || \infty + [A] P))) & \\ S[[]]+\infty P A B Q l m (\text{tnode } tr) = \text{tnode } (S[[]]+p (\text{forcep } P) A B Q l m tr) & \end{aligned}$$

$$\begin{aligned} S[[]]+\infty : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} & \\ \rightarrow (P : \text{Process}\infty \infty c_0) & \\ \rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) & \\ \rightarrow (Q : \text{Process}\infty \infty c_1) & \\ \rightarrow \text{Ref}\infty (P [A] || \infty [B] Q) (\text{fmap}\infty \text{swap} \times ((Q [B] || \infty [A] P))) & \\ S[[]]+\infty P A B Q l m tr = S[[]]+pp ((\text{forcep } P)) A B (\text{forcep } Q) l m tr & \end{aligned}$$

$$\begin{aligned} S[[]]+pp : \{lu : \text{LUniv}\} \{c_0 c_1 : \text{Choice}\} & \\ \rightarrow (P : \text{Process} \infty c_0) & \\ \rightarrow (A B : \text{Label } lu \rightarrow \text{Bool}) & \end{aligned}$$

```

→ (Q : Process ∞ c₁)
→ Ref (P [ A ] [ B ] Q) (fmap swap× ((Q [ B ] [ A ] P)))
S[[]]pp (terminate p) A B (terminate q) .[] .(just (p ,, q)) (ter .(p ,, q)) = ter (p ,, q)
S[[]]pp (terminate p) A B (terminate q) .[] .nothing (empty .(p ,, q)) = empty (p ,, q)
S[[]]pp (terminate p) A B (node q) .[] .nothing (tnode empty) = tnode empty
S[[]]pp (terminate p) A B (node q) .(Lab q a :: l) m (tnode (extc l .m (sub a x) x₁)) =
    tnode (extc l m (sub a x) (lemFmap∞ (λ a₁ → a₁ ,, p)
    swap× (PE (q ↑+ (B \ A)) (sub a x)) l m x₁))
S[[]]pp (terminate p) A B (node q) l m (tnode (intc l .m x x₁)) =
    tnode (intc l m x (lemFmap∞ (λ a → a ,, p)
    swap× (PI (q ↑+ (B \ A)) x) l m x₁))
S[[]]pp (terminate p) A B (node q) .[] .(just (p ,, PT q x)) (tnode (terc x)) = tnode (terc x)
S[[]]pp (node p) A B (terminate q) .[] .nothing (tnode empty) = tnode empty
S[[]]pp (node p) A B (terminate q) .(Lab p a :: l) m (tnode (extc l .m (sub a x) x₁)) =
    tnode (extc l m (sub a x)
    (lemFmap∞ (→,, q) swap×
    (PE (p ↑+ (A \ B)) (sub a x)) l m x₁))
S[[]]pp (node p) A B (terminate q) l m (tnode (intc l .m x x₁)) =
    tnode (intc l m x
    (lemFmap∞ (→,, q) swap× (PI (p ↑+ (A \ B)) x) l m x))
S[[]]pp (node p) A B (terminate q) .[] .(just (PT p x ,, q)) (tnode (terc x)) = tnode (terc x)
S[[]]pp (node p) A B (node q) .[] .nothing (tnode empty) = tnode empty
S[[]]pp (node p) A B (node q) .(Lab q a :: l) m (tnode (extc l .m (inj₁ (inj₁ (sub a x))) x₁)) =
    tnode (extc l m (inj₁ (inj₂ (sub a x)))
    (S[[]]++ p A B (PE q a) l m x₁))
S[[]]pp (node p) A B (node q) .(Lab p a :: l) m (tnode (extc l .m (inj₁ (inj₂ (sub a x))) x₁)) =
    tnode (extc l m (inj₁ (inj₁ (sub a x)))
    (S[[]]∞+ (PE p a) A B q l m x₁))
S[[]]pp {lu} (node p) A B (node q) .(Lab q a :: l) m
    (tnode (extc l .m (inj₂ (sub (x ,, x₁) x₂)) x₃)) = let

    lxlx : T' (Lab q x == Lab p x₁)
    lxlx = lemmaBool (Lab q x == Lab p x₁)
    (B (Lab q x)
    ∧ A (Lab p x₁)) x₂

    BQx : T' (B (Lab q x))
    BQx = lemmaBool (B (Lab q x))
    (A (Lab p x₁))
    (lemmaBoolR
    ((Lab q x == Lab p x₁)))

```

$$(B (\text{Lab } q \ x) \wedge \\ A (\text{Lab } p \ x_1)) \ x_2)$$

$$APx_1 : \text{T}' (A (\text{Lab } p \ x_1)) \\ APx_1 = \text{lemmaBool}' ((\text{Lab } q \ x ==| \text{Lab } p \ x_1)) \\ (B (\text{Lab } q \ x)) \\ (A (\text{Lab } p \ x_1)) \ x_2$$

$$lx_1 lx : \text{T}' (\text{Lab } p \ x_1 ==| \text{Lab } q \ x) \\ lx_1 lx = \text{sym} ==| \{lu\} \{\text{Lab } q \ x\} \{\text{Lab } p \ x_1\} \ lx lx$$

$$x_2' : \text{T}' ((\text{Lab } p \ x_1 ==| \text{Lab } q \ x) \\ \wedge A (\text{Lab } p \ x_1) \\ \wedge B (\text{Lab } q \ x)) \\ x_2' = \text{lemmaBool}'' \\ ((\text{Lab } p \ x_1 ==| \text{Lab } q \ x)) \\ (A (\text{Lab } p \ x_1)) \\ (B (\text{Lab } q \ x)) \\ lx_1 lx APx_1 BQx$$

$$\text{auxproof} : \text{Tr+} (\text{Lab } p \ x_1 :: l) \\ m (p [A] || + [B] q) \\ \text{auxproof} = \text{extc } l \ m (\text{inj}_2 (\text{sub } (x_1 \ \text{.. } x) \\ x_2')) \\ (\text{S}[[]]\infty\infty (\text{PE } p \ x_1) \\ A \ B (\text{PE } q \ x) \ l \ m \ x_3)$$

$$\text{auxproof}' : \text{Tr+} (\text{Lab } q \ x :: l) \ m \\ (p [A] || + [B] q) \\ \text{auxproof}' = \text{transfLu } \{lu\} (\lambda \ l' \rightarrow \text{Tr+} (l' :: l) \\ m (p [A] || + [B] q)) \\ \{\text{Lab } p \ x_1\} \\ \{\text{Lab } q \ x\} \ lx_1 lx \\ \text{auxproof}$$

$$\text{in } (\text{tnode } \text{auxproof}')$$

$$\text{S}[[]]\text{pp } (\text{node } p) \ A \ B (\text{node } q) \ l \ m (\text{tnode } (\text{intc } l \ .m \ (\text{inj}_1 \ x) \ x_1)) = \text{tnode } (\text{intc } l \ m \ (\text{inj}_2 \ x) \ x_1) \\ \text{S}[[]]\text{pp } (\text{node } p) \ A \ B (\text{node } q) \ l \ m (\text{tnode } (\text{intc } l \ .m \ (\text{inj}_2 \ y) \ x_1)) = \text{tnode } ((\text{intc } l \ m \ (\text{inj}_1 \ y) \ x_1))$$

```
S[[]]pp (node p) A B (node q) .[] .(just (PT p x1 ,, PT q x)) (tnode (terc (x ,, x1))) = tnode (terc (x1
```

```
S[[]]+p : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process+ ∞ {lu} c0)
  → (A B : Label lu → Bool)
  → (Q : Process ∞ c1)
  → Ref+ (P [ A ]||+p[ B ] Q) (fmap+ swap× ((Q [ B ]||p+ [ A ] P)))
S[[]]+p P A B (terminate x) l m q = lemFmap+ (—,— x) swap× (P ↑+ (A \ B)) l m q
S[[]]+p P A B (node Q) l m q = S[[]]+ P A B Q l m q
```

```
S[[]]p+ : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process ∞ c0)
  → (A B : Label lu → Bool)
  → (Q : Process+ ∞ {lu} c1)
  → Ref+ (P [ A ]||p+ [ B ] Q)(fmap+ swap× ((Q [ B ]||p+ [ A ] P)))
S[[]]p+ (terminate x) Q l m q = lemFmap+ (λ a → a ,, x) swap× (m ↑+ (l \ Q)) q
S[[]]p+ (node P) Q l m q = S[[]]+ P Q l m q
```

A.90 proofSymParR.agda

```
--@PREFIX@mainproofSymParR
```

```
module proofSymParR where
```

```
open import process
open import Size
open import choiceSetU
open import Data.Maybe
open import Data.List
open import Data.Sum
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import lemFmap
open import auxData
open import parallelSimple
```

```

open import Data.Bool.Base renaming (T to T')
open import restrict
open import auxLemmaPar
open import labelUniv

```

```
mutual
```

```
--@BEGIN@natDef
```

```

S[[]]+R : {lu : LUniv}{c0 c1 : Choice} (P : Process+ ∞ {lu} c0)
          (A B : Label lu → Bool) (Q : Process+ ∞ {lu} c1)
→      fmap+ swap× ((Q [ B ]||+[ A ] P)) ⊑+ (P [ A ]||+[ B ] Q)

```

```

S[[]]+R P A B Q .[] .nothing empty = empty
S[[]]+R P A B Q .(Lab P a :: l) m (extc l .m (inj1 (inj1 (sub a x))) x1) = extc l m (inj1 (inj2
S[[]]+R P A B Q .(Lab Q a :: l) m (extc l .m (inj1 (inj2 (sub a x))) x1) = extc l m (inj1 (inj1
S[[]]+R {lu} P A B Q .(Lab P x :: l) m (extc l .m (inj2 (sub (x ,, x1) x2)) x3) = let
  lxlx1 : T' (Lab P x ==| Lab Q x1)
  lxlx1 = lemmaBool (Lab P x ==| Lab Q x1)
                                     (A (Lab P x) ∧ B (Lab Q x1)) x2

```

```

BQx : T' (B (Lab Q x1))
BQx = lemmaBool' (Lab P x ==| Lab Q x1)
                                     (A (Lab P x) (B (Lab Q x1))) x2

```

```

APx1 : T' (A (Lab P x))
APx1 = lemmaBool (A (Lab P x) (B (Lab Q x1)))
          (lemmaBoolR (Lab P x ==| Lab Q x1)
            (A (Lab P x) ∧ B (Lab Q x1)) x2)

```

```

lx1lx : T' (Lab Q x1 ==| Lab P x)
lx1lx = sym==| {lu} {Lab P x} {Lab Q x1} lxlx1

```

```

x2' : T' ((Lab Q x1 ==| Lab P x) ∧ B (Lab Q x1) ∧ A (Lab P x))
x2' = lemmaBool'' ((Lab Q x1 ==| Lab P x)
  (B (Lab Q x1)) (A (Lab P x)) lx1lx BQx APx1)

```

```

auxproof : Tr+ (Lab Q x1 :: l) m
          (fmap+ swap× (Q [ B ]||+[ A ] P))
auxproof = extc l m (inj2 (sub (x1 ,, x) x2'))
          (S[[]]∞∞R (PE P x) A B (PE Q x1) l m x3)

```



```

auxproof' : Tr+ (Lab P x :: l) m
              (fmap+ swap× (Q [ B ]||+[ A ] P))
auxproof' = transfLu {lu} (λ l' → Tr+ (l' :: l) m
              (fmap+ swap× (Q [ B ]||+[ A ] P)))
              {Lab Q x₁} {Lab P x} lx₁lx auxproof

```

```

                                     in auxproof'
S[[]]+R P A B Q l m (intc .l .m (inj₁ x) x₁) = intc l m (inj₂ x) (S[[]]∞+R (PI P x) A B Q l m x₁)
S[[]]+R P A B Q l m (intc .l .m (inj₂ y) x₁) = intc l m (inj₁ y) (S[[]]∞+R P A B (PI Q y) l m x₁)
S[[]]+R P A B Q .[] .(just (PT P x ,, PT Q x₁)) (terc (x ,, x₁)) = terc ( (x₁ ,, x) )

```

--@END

```

S[[]]∞+R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process+ ∞ {lu} c₀)
  → (A      B : Label lu → Bool)
  → (Q : Process∞ ∞ {lu} c₁)
  → Ref∞ (fmap∞ swap× ((Q [ B ]||∞+[ A ] P))) (P [ A ]||∞+[ B ] Q)
S[[]]∞+R P A B Q l m (tnode tr) = tnode (S[[]]p+R P A B (forcep Q) l m tr)

```

```

S[[]]∞+R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process∞ ∞ {lu} c₀)
  → (A      B : Label lu → Bool)
  → (Q : Process+ ∞ {lu} c₁)
  → Ref∞ (fmap∞ swap× ((Q [ B ]||∞+[ A ] P))) (P [ A ]||∞+[ B ] Q)
S[[]]∞+R P A B Q l m (tnode tr) = tnode (S[[]]p+R (forcep P) A B Q l m tr)

```

```

S[[]]p+R : {lu : LUniv}{c₀ c₁ : Choice}
  → (P : Process+ ∞ {lu} c₀)
  → (A      B : Label lu → Bool)
  → (Q : Process      ∞ {lu} c₁)
  → Ref+ (fmap+ swap× ((Q [ B ]||p+[ A ] P))) (P [ A ]||p+[ B ] Q)
S[[]]p+R P A B (terminate x) l m q = lemFmap+R (↪,↪ x) swap× (P ↗+ (A \ B)) l m q
S[[]]p+R P A B (node Q) l m q = S[[]]+R P A B Q l m q

```



```

S[[]]p+R : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process ∞ {lu} c0)
  → (A      B : Label lu → Bool)
  → (Q : Process+ ∞ {lu} c1)
  → Ref+ (fmap+ swap× ((Q [ B ]||+p[ A ] P))) (P [ A ]||p+ [ B ] Q)
S[[]]p+R (terminate x) Q l m q = lemFmap+R (λ a → a ,, x) swap× (m |+ (l \ Q)) q
S[[]]p+R (node P) Q l m q = S[[]]R P Q l m q

```

```

S[[]]∞∞R : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process∞ ∞ {lu} c0)
  → (A      B : Label lu → Bool)
  → (Q : Process∞ ∞ {lu} c1)
  → Ref∞ (fmap∞ swap× ((Q [ B ]||∞ [ A ] P))) (P [ A ]||∞ [ B ] Q)
S[[]]∞∞R P A B Q l m x = S[[]]ppR ((forcep P)) A B (forcep Q) l m x

```

```

S[[]]ppR : {lu : LUniv}{c0 c1 : Choice}
  → (P : Process ∞ {lu} c0)
  → (A      B : Label lu → Bool)
  → (Q : Process      ∞ {lu} c1)
  → Ref      (fmap swap× ((Q [ B ]|| [ A ] P))) (P [ A ]|| [ B ] Q)
S[[]]ppR (terminate x) A B (terminate x1) .[] .(just (x ,, x1)) (ter .(x ,, x1)) = ter (x ,, x1)
S[[]]ppR (terminate x) A B (terminate x1) .[] .nothing (empty .(x ,, x1)) = empty (x ,, x1)
S[[]]ppR (terminate x) A B (node x1) .[] .nothing (tnode empty) = tnode empty
S[[]]ppR (terminate x) A B (node x1) .(Lab x1 a :: l) m (tnode (extc l .m (sub a x2) x3)) =
  tnode (extc l m (sub a x2)
    (lemFmap∞R (λ a1 → a1 ,, x) swap× (PE (x1 |+ (B \ A)) (sub a x2)))
S[[]]ppR (terminate x) A B (node x1) l m (tnode (intc l .m x2 x3)) =
  tnode (intc l m x2
    (lemFmap∞R (λ a → a ,, x) swap× (PI (x1 |+ (B \ A)) x2 l m x3)))
S[[]]ppR (terminate x) A B (node x1) .[] .(just (x ,, PT x1 x2)) (tnode (terc x2)) = tnode (terc x2)
S[[]]ppR (node x) A B (terminate x1) .[] .nothing (tnode empty) = tnode empty
S[[]]ppR (node x) A B (terminate x1) .(Lab x a :: l) m (tnode (extc l .m (sub a x2) x3)) = tnode
  (lemFmap∞R (λ a → a ,, x) swap× (PE (x |+ (A \ B)) (sub a x2)) l m x3)
S[[]]ppR (node x) A B (terminate x1) l m (tnode (intc l .m x2 x3)) = tnode (intc l m x2
  (lemFmap∞R (λ a → a ,, x) swap× (PI (x |+ (A \ B)) x2 l m x3)))
S[[]]ppR (node x) A B (terminate x1) .[] .(just (PT x x2 ,, x1)) (tnode (terc x2)) = tnode (terc x2)
S[[]]ppR (node x) A B (node x1) .[] .nothing (tnode empty) = tnode empty
S[[]]ppR (node x) A B (node x1) .(Lab x a :: l) m (tnode (extc l .m (inj1 (inj1 (sub a x2)))) x3) =
  tnode (extc l m (inj1 (inj2 (sub a x2))) (S[[]]∞+R (PE x a) A B x1 l m x3))
S[[]]ppR (node x) A B (node x1) .(Lab x1 a :: l) m (tnode (extc l .m (inj1 (inj2 (sub a x2)))) x3) =

```

```

      tnode (extc l m (inj1 (inj1 (sub a x2))) (S[[]]+∞R x A B (PE x1 a) l m x3))
S[[]]ppR {lu} (node P) A B (node Q) .(Lab P x :: l) m (tnode (extc l .m (inj2 (sub (x ,, x1) x2)) x3))
  lxlx1 : T' (Lab P x ==| Lab Q x1)
  lxlx1 = lemmaBool (Lab P x ==| Lab Q x1)
                                (A (Lab P x) ∧ B (Lab Q x1)) x2

BQx : T' (B (Lab Q x1))
BQx = lemmaBool' (Lab P x ==| Lab Q x1)
                                (A (Lab P x)) (B (Lab Q x1)) x2

APx1 : T' (A (Lab P x))
APx1 = lemmaBool (A (Lab P x)) (B (Lab Q x1))
      (lemmaBoolR (Lab P x ==| Lab Q x1)
        (A (Lab P x) ∧ B (Lab Q x1)) x2)

lx1lx : T' (Lab Q x1 ==| Lab P x)
lx1lx = sym==| {lu} {Lab P x} {Lab Q x1} lxlx1

x2' : T' ((Lab Q x1 ==| Lab P x) ∧ B (Lab Q x1) ∧ A (Lab P x))
x2' = lemmaBool'' ((Lab Q x1 ==| Lab P x))
      (B (Lab Q x1)) (A (Lab P x)) lx1lx BQx APx1

auxproof : Tr+ (Lab Q x1 :: l) m
      (fmap+ swap× (Q [ B ]||+[ A ] P))
auxproof = extc l m (inj2 (sub (x1 ,, x) x2'))
      (S[[]]∞∞R (PE P x) A B (PE Q x1) l m x3)

auxproof' : Tr+ (Lab P x :: l) m
      (fmap+ swap× (Q [ B ]||+[ A ] P))
auxproof' = transfLu {lu} (λ l' → Tr+ (l' :: l) m
      (fmap+ swap× (Q [ B ]||+[ A ] P))
      {Lab Q x1} {Lab P x} lx1lx auxproof

      in tnode auxproof'
S[[]]ppR (node x) A B (node x1) l m (tnode (intc .l .m (inj1 x2) x3)) = tnode (intc l m (inj2 x2) (S[[]]c
S[[]]ppR (node x) A B (node x1) l m (tnode (intc .l .m (inj2 y) x3)) = tnode (intc l m (inj1 y) (S[[]]c
S[[]]ppR (node x) A B (node x1) .[] .(just (PT x x2 ,, PT x1 x3)) (tnode (terc (x2 ,, x3))) = tnode (terc

```

A.91 proofTerHide.agda

```
--@PREFIX@mainproofTerHide
```

```
module proofTerHide where
```

```
open import process
open import Size
open import choiceSetU
open import auxData
open import Data.Maybe
open import Data.List
open import Data.Sum
open import labelUniv
open import dataAuxFunction
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import hidingOperator
open import primitiveProcess
open import Data.Bool
open import traceEquivalence
open import Data.Product
```

```
--@BEGIN@unitHideLaw
```

```
unitHideLaw : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c0)
  → Hide A (terminate a) ⊆ (terminate a)
unitHideLaw {i} {c0} a A P l m q = q
```

```
unitHideLawr : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c0)
  → (terminate a) ⊆ Hide A (terminate a)
unitHideLawr {i} {c0} a A P l m q = q
```

```
--@END
```

```
--@BEGIN@unitHideLawTheo
```

```
≡unitHide : {i : Size} {lu : LUniv} {c₀ : Choice} (a : ChoiceSet c₀)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c₀)
  → Hide A (terminate a) ≡ (terminate a)
```

```
--@END
```

```
--@BEGIN@unitHideLawTheoProof
```

```
≡unitHide a A P = (unitHideLaw a A P) , (unitHideLawr a A P)
```

```
--@END
```

```
--@BEGIN@stopHideLaw
```

```
stopHideLaw : {i : Size} {lu : LUniv} {c₀ : Choice} (a : ChoiceSet c₀)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c₀)
  → Hide A (STOP c₀) ⊆ ((STOP c₀))
stopHideLaw {i} {c₀} a A (terminate x) .[] .nothing (tnode empty)
  = tnode empty
stopHideLaw {i} {c₀} a A (terminate x₁) .(efq _ :: l) m
  (tnode (extc l .m () x₂))
stopHideLaw {i} {c₀} a A (terminate x) l m
  (tnode (intc .l .m () x₂))
stopHideLaw {i} {c₀} a A (terminate x₁) .[] .(just (efq x))
  (tnode (terc x)) = tnode (terc x)
stopHideLaw {i} {c₀} a A (node x) .[] .nothing (tnode empty)
  = tnode empty
stopHideLaw {i} {c₀} a A (node x₁) .(efq _ :: l) m
  (tnode (extc l .m () x₂))
stopHideLaw {i} {c₀} a A (node x) l m (tnode (intc .l .m () x₂))
stopHideLaw {i} {c₀} a A (node x₁) .[] .(just (efq x))
  (tnode (terc x)) = tnode (terc x)
```

--@END

```

stopHideLawr : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c0)
  → (STOP c0)  $\sqsubseteq$  Hide A (STOP c0)
stopHideLawr {i} {c0} a A (terminate x) .[] .nothing (tnode empty) = tnode empty
stopHideLawr {i} {c0} a A (terminate x1) .(efq - :: l) m (tnode (extc l .m (sub () x) x2))
stopHideLawr {i} {c0} a A (terminate x) l m (tnode (intc .l .m (inj1 ()) x2))
stopHideLawr {i} {c0} a A (terminate x) l m (tnode (intc .l .m (inj2 (sub () x1)) x2))
stopHideLawr {i} {c0} a A (terminate x1) .[] .(just (efq x)) (tnode (terc x)) = tnode (terc x)
stopHideLawr {i} {c0} a A (node x) .[] .nothing (tnode empty) = tnode empty
stopHideLawr {i} {c0} a A (node x1) .(efq - :: l) m (tnode (extc l .m (sub () x) x2))
stopHideLawr {i} {c0} a A (node x) l m (tnode (intc .l .m (inj1 ()) x2))
stopHideLawr {i} {c0} a A (node x) l m (tnode (intc .l .m (inj2 (sub () x1)) x2))
stopHideLawr {i} {c0} a A (node x1) .[] .(just (efq x)) (tnode (terc x)) = tnode (terc x)

```

--@BEGIN@stopHideLawEq

```

 $\equiv$ stopHide : {i : Size} {lu : LUniv} {c0 : Choice} (a : ChoiceSet c0)
  → (A : Label lu → Bool)
  → (P : Process i {lu} c0)
  → Hide A (STOP c0)  $\equiv$  (STOP c0)

```

--@END

--@BEGIN@stopHideLawEqProof

```

 $\equiv$ stopHide a A P = (stopHideLaw a A P) , (stopHideLawr a A P)

```

--@END

A.92 proofTerInter.agda

--@PREFIX@mainproofTerInter

```
module proofTerInter where
```

```
open import process
open import Size
open import choiceSetU
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import auxData
open import label
open import parallelSimple
open import restrict
open import Data.Bool
open import interleaved
open import traceEquivalence
open import Data.Product
open import labelUniv
```

```
--@BEGIN@TerIntLaw
```

```
TerIntLaw : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0) (b : ChoiceSet c1)
           →  $\sqsubseteq$  {lu} (terminate a ||| terminate b) (terminate ((a ,, b)))
TerIntLaw {lu}{c0} {c1} a P l m q = q
```

```
TerIntLawr : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0) (b : ChoiceSet c1)
           →  $\sqsubseteq$  {lu} (terminate ((a ,, b))) (terminate a ||| terminate b)
TerIntLawr {lu} {c0} {c1} a P l m q = q
```

```
--@END
```

```
--@BEGIN@TerIntLawTheo
```

```
 $\equiv$ TerInt+ : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0) (b : ChoiceSet c1)
           →  $\equiv$  {lu} (terminate a ||| terminate b) (terminate ((a ,, b)))
```

```
--@END
```

```

--@BEGIN@TerIntLawTheoProof

≡TerInt+ a b = TerIntLaw a b , TerIntLawr a b

--@END

--@BEGIN@uniIntLaw

uniIntLaw : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
           → (P : Process ∞ {lu} c₀)
           → (terminate a ||| P) ⊆ fmap ((_,_) a)) P
uniIntLaw {c₀} a P l m q = q

uniIntLawr : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
           → (P : Process ∞ {lu} c₀)
           → fmap ((_,_) a)) P ⊆ (terminate a ||| P)
uniIntLawr {c₀} a P l m q = q

--@END

--@BEGIN@uniIntLawTheo

≡uniInt : {lu : LUniv}{c₀ : Choice} (a : ChoiceSet c₀)
        → (P : Process ∞ {lu} c₀)
        → (terminate a ||| P) ≡ fmap ((_,_) a)) P

--@END

--@BEGIN@uniIntLawTheoProof

≡uniInt a P = (uniIntLaw a P) , (uniIntLawr a P)

--@END

```

A.93 proofTerPar.agda

```

--@PREFIX@mainproofTerPar

```

```
module proofTerPar where
```

```
open import process
open import Size
open import choiceSetU
open import renamingResult
open import TraceWithoutSize
open import RefWithoutSize
open import auxData
open import parallelSimple
open import restrict
open import Data.Bool
open import traceEquivalence
open import Data.Product
open import labelUniv
```

```
--@BEGIN@unitPar
```

```
unit[-|-] : {lu : LUniv}{c0 : Choice} (a : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → (terminate a [ A ]||[ B ] P) ⊆ fmap ( (λ b → (a „ b))) (P ↑ (B \ A))
unit[-|-] {c0} a P A B l m q = q
```

```
unit[-|-]r : {lu : LUniv}{c0 : Choice} (a : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → fmap ( (λ b → (a „ b))) (P ↑ (B \ A)) ⊆ (terminate a [ A ]||[ B ] P)
unit[-|-]r {c0} a P A B l m q = q
```

```
≡U+ : {lu : LUniv}{c0 : Choice} (a : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → (terminate a [ A ]||[ B ] P) ≡ fmap ( (λ b → (a „ b))) (P ↑ (B \ A))
≡U+ a P A B = (unit[-|-] a P A B) , (unit[-|-]r a P A B)
```

```
--@END
```

```
unit[-|-]R : {lu : LUniv}{c0 : Choice} (b : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → (P [ B ]||| [ A ] terminate b) ⊆ fmap (λ a → (a „ b))(P ↑ (B \ A))
```

```
unit[-|-]R      {c0} a (terminate x) A B l m q = q
unit[-|-]R      {c0} a (node x) A B l m (tnode x1) = tnode x1
```

```
unit[-|-]RR : {lu : LUniv}{c0 : Choice} (b : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → fmap (λ a → (a „ b))(P ↑ (B \ A)) ⊆ (P [ B ]||| [ A ] terminate b)
```

```
unit[-|-]RR {c0} a (terminate x) A B l m q = q
unit[-|-]RR {c0} a (node x) A B l m (tnode x1) = tnode x1
```

```
≡Ur+ : {lu : LUniv}{c0 : Choice} (b : ChoiceSet c0)(P : Process ∞ c0) (A B : Label lu → Bool)
  → (P [ B ]||| [ A ] terminate b) ≡ fmap (λ a → (a „ b))(P ↑ (B \ A))
≡Ur+ b P A B = (unit[-|-]R b P A B) , (unit[-|-]RR b P A B)
```

```
--@END
```

```
--@BEGIN@terPr
```

```
ter[-|-] : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (b : ChoiceSet c1) (A B : Label lu → Bool)
  → (terminate a [ A ]||| [ B ] terminate b) ⊆
    fmap (λ x → a „ b) (terminate ((a „ b)))
ter[-|-] {c0} a P A B l m q = q
```

```
ter[-|-]r : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (b : ChoiceSet c1) (A B : Label lu → Bool)
  → fmap (λ x → a „ b) (terminate ((a „ b))) ⊆
    (terminate a [ A ]||| [ B ] terminate b)
ter[-|-]r a P A B l m q = q
```

```
--@END
```

```
--@BEGIN@terPrTheo
```

```
≡ter[-|-] : {lu : LUniv}{c0 c1 : Choice} (a : ChoiceSet c0)
  (b : ChoiceSet c1) (A B : Label lu → Bool)
```

$$\rightarrow (\text{terminate } a \text{ [} A \text{]} \text{ [} B \text{] } \text{terminate } b)$$

$$\equiv \text{fmap } (\lambda x \rightarrow a \text{ ,, } b) (\text{terminate } ((a \text{ ,, } b)))$$

```
--@END
```

```
--@BEGIN@terPrTheoProof
```

```
 $\equiv \text{ter}[-||-] \ a \ b \ A \ B = (\text{ter}[-||-] \ a \ b \ A \ B) \ , \ (\text{ter}[-||-] \text{r} \ a \ b \ A \ B)$ 
```

```
--@END
```

A.94 rec.agda

```
--@PREFIX@Rec
```

```
module rec where
```

```
open import Data.String renaming (_++_ to _++s_)
```

```
open import Data.Sum
```

```
open import Size
```

```
open import choiceSetU
```

```
open import process
```

```
open import dataAuxFunction
```

```
open import sequentialComposition
```

```
open import showFunction
```

```
open import renamingProcess
```

```
open import labelUniv
```

```
--@BEGIN@recDef
```

```
mutual
```

```
rec : {i : Size} → {lu : LUniv}{c0 c1 : Choice}
      → (s : String)
      → (ChoiceSet c0 → Process+ (↑ i) {lu} (c0  $\uplus$  c1))
      → ChoiceSet c0
      → Process $\infty$  i {lu} c1
```

```
forcep (rec s f a) = renameP s
                      (f a  $\gg$ ==+p recaux s f)
```

```
Str $\infty$  (rec s f a) = s
```

```
recaux : {i : Size} → {lu : LUniv}{c0 c1 : Choice}
        → (s : String)
```

```

      → (ChoiceSet c₀ → Process+ (↑ i) {lu} (c₀ ⊕' c₁))
      → (ChoiceSet c₀ ⊕ ChoiceSet c₁)
      → Process∞ i {lu} c₁
recaux s f (inj₁ x) = rec s f x
recaux s f (inj₂ x) = delay (terminate x)

recStr : {lu : LUniv}{c₀ : Choice}
        → (ChoiceSet c₀ → String)
        → ChoiceSet c₀ → String
recStr f a = "rec(" ++ choice2Str2Str f ++s ", " ++s choice2Str a ++s ")"

--@END

recAutoStr : {i : Size} → {lu : LUniv}{c₀ c₁ : Choice}
            → (ChoiceSet c₀ → Process+ (↑ i) {lu} (c₀ ⊕' c₁))
            → ChoiceSet c₀
            → Process∞ i c₁
recAutoStr {i} {lu} f a = rec (recStr {lu} (Str+ ∘ f) a) f a

```

A.95 RefWithoutSize.agda

```
--@PREFIX@mainRefWithoutSize
```

```
module RefWithoutSize where
```

```

open import Size
open import Data.List
open import Data.Product
open import Data.Maybe
open import labelUniv
open import process
open import choiceSetU
open import TraceWithoutSize

```

```
--@BEGIN@refDef
```

```

 $\_ \sqsubseteq \_$  : {lu : LUniv}{c : Choice} (P : Process  $\infty$  {lu} c)
      (Q : Process  $\infty$  {lu} c)  $\rightarrow$  Set
 $\_ \sqsubseteq \_$  {lu} {c} P Q = (l : List (Label lu))
       $\rightarrow$  (m : Maybe (ChoiceSet c))
       $\rightarrow$  (tr : Tr {lu} l m Q)  $\rightarrow$  Tr {lu} l m P

--@END

--@BEGIN@refInfDef

 $\_ \sqsubseteq \infty \_$  : {lu : LUniv}{c : Choice} (P : Process $\infty$   $\infty$  {lu} c)
      (Q : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  Set
 $\_ \sqsubseteq \infty \_$  {lu} {c} P Q = (l : List (Label lu))
       $\rightarrow$  (m : Maybe (ChoiceSet c))
       $\rightarrow$  (tr : Tr $\infty$  {lu} l m Q)  $\rightarrow$  Tr $\infty$  {lu} l m P

--@END

--@BEGIN@refPlusDef

 $\_ \sqsubseteq + \_$  : {lu : LUniv}{c : Choice} (P : Process+  $\infty$  {lu} c)
      (Q : Process+  $\infty$  {lu} c)  $\rightarrow$  Set
 $\_ \sqsubseteq + \_$  {lu}{c} P Q = (l : List (Label lu))
       $\rightarrow$  (m : Maybe (ChoiceSet c))
       $\rightarrow$  (tr : Tr+ {lu} l m Q)  $\rightarrow$  Tr+ {lu} l m P

--@END

--@BEGIN@natDef

Ref =  $\_ \sqsubseteq \_$ 
Ref $\infty$  =  $\_ \sqsubseteq \infty \_$ 
Ref+ =  $\_ \sqsubseteq + \_$ 

```

```

 $\_ \sqsubseteq_r \_$  : {lu : LUniv}{c c1 : Choice} (P : Process  $\infty$  {lu} c)
      (Q : Process  $\infty$  {lu} c1)  $\rightarrow$  Set

```

```


$$\begin{aligned}
\text{--}\sqsubseteq_r\text{--} \quad \{lu\}\{c\} \{c_1\} P Q &= (l : \text{List (Label } lu)) \\
&\rightarrow (m : \text{Maybe (ChoiceSet } c)) \\
&\rightarrow (m_1 : \text{Maybe (ChoiceSet } c_1)) \\
&\rightarrow \text{Tr } \{lu\} \, l \, m_1 \, Q \rightarrow \text{Tr } \{lu\} \, l \, m \, P \\
\text{Ref}_r &= \text{--}\sqsubseteq_r\text{--} \\
\text{--}\sqsubseteq_{\infty_r}\text{--} : \{lu : \text{LUniv}\} \{c \, c_1 : \text{Choice}\} (P : \text{Process}_{\infty} \, \infty \, \{lu\} \, c) \\
&\quad (Q : \text{Process}_{\infty} \, \infty \, \{lu\} \, c_1) \rightarrow \text{Set} \\
\text{--}\sqsubseteq_{\infty_r}\text{--} \quad \{lu\} \{c\} \{c_1\} P Q &= (l : \text{List (Label } lu)) \\
&\rightarrow (m : \text{Maybe (ChoiceSet } c)) \\
&\rightarrow (m_1 : \text{Maybe (ChoiceSet } c_1)) \\
&\rightarrow \text{Tr}_{\infty} \, \{lu\} \, l \, m_1 \, Q \rightarrow \text{Tr}_{\infty} \, \{lu\} \, l \, m \, P \\
\text{Ref}_{\infty_r} &= \text{--}\sqsubseteq_{\infty_r}\text{--} \\
\text{--}\sqsubseteq_{+r}\text{--} : \{lu : \text{LUniv}\} \{c \, c_1 : \text{Choice}\} (P : \text{Process}_{+} \, \infty \, \{lu\} \, c) \\
&\quad (Q : \text{Process}_{+} \, \infty \, \{lu\} \, c_1) \rightarrow \text{Set} \\
\text{--}\sqsubseteq_{+r}\text{--} \quad \{lu\} \{c\} \{c_1\} P Q &= (l : \text{List (Label } lu)) \\
&\rightarrow (m : \text{Maybe (ChoiceSet } c)) \\
&\rightarrow (m_1 : \text{Maybe (ChoiceSet } c_1)) \\
&\rightarrow \text{Tr}_{+} \, \{lu\} \, l \, m_1 \, Q \rightarrow \text{Tr}_{+} \, \{lu\} \, l \, m \, P \\
\text{Ref}_{+r} &= \text{--}\sqsubseteq_{+r}\text{--} \\
\text{--@END}
\end{aligned}$$


```

A.96 renamingOperator.agda

```
--@PREFIX@renamingOperator
```

```
module renamingOperator where
```

```

open import Size
open import process
open import choiceSetU
open import choiceAuxFunction
open import dataAuxFunction
open import auxData
open import labelUniv
open import Data.String renaming (_++_ to _++s_)
open import showLabelP hiding (labelLabelFunToString)

```

```
--@BEGIN@renamingOperatorDef
```

```
RenameStr : {lu : LUniv}(f : Label lu → Label lu) → String → String
RenameStr f s = "(" ++ s ++ ")" ++ (labelLabelFunToString f)
```

```
mutual
```

```
Rename∞ : {i : Size} → {lu : LUniv} {c : Choice}
          → (f : Label lu → Label lu)
          → Process∞ i {lu} c → Process∞ i {lu} c
```

```
forcep (Rename∞ f P) = Rename f (forcep P)
Str∞ (Rename∞ f P) = RenameStr f (Str∞ P)
```

```
Rename : {i : Size} → {lu : LUniv} {c : Choice}
         → (f : Label lu → Label lu)
         → Process i {lu} c → Process i {lu} c
```

```
Rename f (node P) = node (Rename+ f P)
Rename f (terminate x) = terminate x
```

```
Rename+ : {i : Size} → {lu : LUniv} {c : Choice}
          → (f : Label lu → Label lu)
          → Process+ i {lu} c → Process+ i {lu} c
```

```
E (Rename+ f P) = (E P)
Lab (Rename+ f P) c = f (Lab P c)
PE (Rename+ f P) c = Rename∞ f (PE P c)
I (Rename+ f P) = I P
PI (Rename+ f P) c = Rename∞ f (PI P c)
T (Rename+ f P) = T P
PT (Rename+ f P) c = PT P c
Str+ (Rename+ f P) = RenameStr f (Str+ P)
```

```
--@END
```

```
mutual
```

```
RenameWithName∞ : {i : Size}
                  → {c : Choice}
                  → {lu : LUniv}
                  → (name : String → String)
                  → (f : Label lu → Label lu)
                  → Process∞ i {lu} c
                  → Process∞ i {lu} c
```

```
forcep (RenameWithName∞ name f P) = RenameWithName name f (forcep P)
```

$\text{Str}\infty (\text{RenameWithName}\infty \text{ name } f P) = \text{name } (\text{Str}\infty P)$

$\text{RenameWithName} : \{i : \text{Size}\}$
 $\rightarrow \{c : \text{Choice}\}$
 $\rightarrow \{lu : \text{LUniv}\}$
 $\rightarrow (\text{name} : \text{String} \rightarrow \text{String})$
 $\rightarrow (f : \text{Label } lu \rightarrow \text{Label } lu)$
 $\rightarrow \text{Process } i \{lu\} c$
 $\rightarrow \text{Process } i \{lu\} c$

$\text{RenameWithName } \text{ name } f (\text{node } P) = \text{node } (\text{RenameWithName+ } \text{ name } f P)$

$\text{RenameWithName } \text{ name } f (\text{terminate } x) = \text{terminate } x$

$\text{RenameWithName+} : \{i : \text{Size}\}$
 $\rightarrow \{c : \text{Choice}\}$
 $\rightarrow \{lu : \text{LUniv}\}$
 $\rightarrow (\text{name} : \text{String} \rightarrow \text{String})$
 $\rightarrow (f : \text{Label } lu \rightarrow \text{Label } lu)$
 $\rightarrow \text{Process+ } i \{lu\} c$
 $\rightarrow \text{Process+ } i \{lu\} c$

$\text{E } (\text{RenameWithName+ } \text{ name } f P) = (\text{E } P)$

$\text{Lab } (\text{RenameWithName+ } \text{ name } f P) c = f (\text{Lab } P c)$

$\text{PE } (\text{RenameWithName+ } \text{ name } f P) c = \text{RenameWithName}\infty \text{ name } f (\text{PE } P c)$

$\text{I } (\text{RenameWithName+ } \text{ name } f P) = \text{I } P$

$\text{PI } (\text{RenameWithName+ } \text{ name } f P) c = \text{RenameWithName}\infty \text{ name } f (\text{PI } P c)$

$\text{T } (\text{RenameWithName+ } \text{ name } f P) = \text{T } P$

$\text{PT } (\text{RenameWithName+ } \text{ name } f P) c = \text{PT } P c$

$\text{Str+ } (\text{RenameWithName+ } \text{ name } f P) = \text{name } (\text{Str+ } P)$

A.97 renamingProcess.agda

module renamingProcess where

open import Data.String

open import Size

open import process

open import choiceSetU

open import labelUniv

mutual

```
renameP∞ : {i : Size} → {lu : LUniv}{c : Choice} → String → Process∞ i {lu} c → Process∞ i {
forceP (renameP∞ {i} s P) {j} = renameP {j} s (forceP P {j})
Str∞ (renameP∞ {i} s P) = s
```

```
renameP : {i : Size} → {lu : LUniv}{c : Choice} → String → Process i {lu} c
        → Process i {lu} c
renameP s (node P) = node (renameP+ s P)
renameP s (terminate x) = terminate x
```

```
renameP+ : {i : Size} → {lu : LUniv}{c : Choice} → String → Process+ i {lu} c
        → Process+ i c
E (renameP+ s P) = (E P)
Lab (renameP+ s P) c = Lab P c
PE (renameP+ s P) c = PE P c
I (renameP+ s P) = I P
PI (renameP+ s P) c = PI P c
T (renameP+ s P) = T P
PT (renameP+ s P) c = PT P c
Str+ (renameP+ s P) = s
```

A.98 renamingResult.agda

```
--@PREFIX@renamingResult
```

```
module renamingResult where
```

```
open import process
open import Size
open import choiceSetU
open import sequentialComposition
open import dataAuxFunction
open import Data.String renaming (_++_ to _++s_)
open import showFunctionForSimulator
open import labelUniv
```

```
--@BEGIN@renamingDef
```

```
fmapStr : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)
        → String → String
fmapStr f str = "(fmap " ++s choiceFunToStr↓ f ++s " " ++s str ++s ")"
```

mutual

```
fmap∞ : {c0 c1 : Choice}
       → (f : ChoiceSet c0 → ChoiceSet c1)
       → {i : Size}
       → {lu : LUniv}
       → Process∞ i {lu} c0 → Process∞ i {lu} c1
forcep (fmap∞ f P)      = fmap f (forcep P)
Str∞   (fmap∞ f P)      = fmapStr f (Str∞ P)
```

```
fmap : {lu : LUniv}{c0 c1 : Choice} → (f : ChoiceSet c0
    → ChoiceSet c1) → {i : Size}
    → Process i {lu} c0 → Process i {lu} c1
fmap f (terminate a) = terminate (f a)
fmap f (node P)      = node (fmap+ f P)
```

```
fmap+ : {c0 c1 : Choice}
       → (f : ChoiceSet c0 → ChoiceSet c1)
       → {i : Size}
       → {lu : LUniv}
       → Process+ i {lu} c0 → Process+ i {lu} c1
E   (fmap+ f P)      = E P
Lab (fmap+ f P) c    = Lab P c
PE  (fmap+ f P) c    = fmap∞ f (PE P c)
I   (fmap+ f P)      = I P
PI  (fmap+ f P) c    = fmap∞ f (PI P c)
T   (fmap+ f P)      = T P
PT  (fmap+ f P) c    = f (PT P c)
Str+ (fmap+ f P)     = fmapStr f (Str+ P)
```

--@END

```
fmapi : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → (i : Size)
    → {lu : LUniv}
```

```

      → Process i {lu} c0
      → Process i {lu} c1
fmapi f i P = fmap f {i} P

```

```

fmap+i : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → (i : Size)
      → {lu : LUniv}
      → Process+ i {lu} c0
      → Process+ i {lu} c1
fmap+i f i P = fmap+ f {i} P

```

```

fmap∞i : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → (i : Size)
      → {lu : LUniv}
      → Process∞ i {lu} c0
      → Process∞ i {lu} c1
fmap∞i f i = fmap∞ f {i}

```

```

fmap' : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → {i : Size}
      → {lu : LUniv}
      → Process i {lu} c0
      → Process i {lu} c1
fmap' f P = P >>= (delay ∘ (terminate ∘ f))

```

```

fmap+' : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → {i : Size}
      → {lu : LUniv}
      → Process+ i {lu} c0
      → Process+ i {lu} c1
fmap+' f P = P >>=+ (delay ∘ (terminate ∘ f))

```

```

fmap∞' : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1) → {i : Size}
      → {lu : LUniv}
      → Process∞ i {lu} c0
      → Process∞ i {lu} c1
fmap∞' f P = P >>=∞ (delay ∘ (terminate ∘ f))

```

mutual

```

fmapCustomStr : {c0 c1 : Choice} → (f : ChoiceSet c0 → ChoiceSet c1)

```

```

→ (fname : String)
→ String → String
fmapCustomStr f fname str = "(fmap " ++s fname ++s " " ++s str ++s ")"

```

```

fmapCustom∞ : {c₀ c₁ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁) → {i : Size}
→ {lu : LUniv}
→ (fname : String)
→ Process∞ i {lu} c₀ → Process∞ i {lu} c₁
forcep (fmapCustom∞ f fname P) = fmapCustom f fname (forcep P)
Str∞ (fmapCustom∞ f fname P) = fmapCustomStr f fname (Str∞ P)

```

```

fmapCustom : {c₀ c₁ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁) → {i : Size}
→ {lu : LUniv}
→ (fname : String)
→ Process i {lu} c₀
→ Process i {lu} c₁
fmapCustom f fname (terminate a) = terminate (f a)
fmapCustom f fname (node P) = node (fmapCustom+ f fname P)

```

```

fmapCustom+ : {c₀ c₁ : Choice} → (f : ChoiceSet c₀ → ChoiceSet c₁) → {i : Size}
→ {lu : LUniv}
→ (fname : String)
→ Process+ i {lu} c₀
→ Process+ i {lu} c₁
E (fmapCustom+ f fname P) = E P
Lab (fmapCustom+ f fname P) c = Lab P c
PE (fmapCustom+ f fname P) c = fmapCustom∞ f fname (PE P c)
I (fmapCustom+ f fname P) = I P
PI (fmapCustom+ f fname P) c = fmapCustom∞ f fname (PI P c)
T (fmapCustom+ f fname P) = T P
PT (fmapCustom+ f fname P) c = f (PT P c)
Str+ (fmapCustom+ f fname P) = fmapCustomStr f fname (Str+ P)

```

mutual

```

fmapWithName∞ : {c₀ c₁ : Choice}
→ (name : String → String)
→ (f : ChoiceSet c₀ → ChoiceSet c₁)
→ {i : Size}
→ {lu : LUniv}

```

```

      → Process∞ i {lu} c₀ → Process∞ i {lu} c₁
forcep (fmapWithName∞ name f P) = fmapWithName name f (forcep P)
Str∞ (fmapWithName∞ name f P) = name (Str∞ P)

```

```

fmapWithName : {c₀ c₁ : Choice}
              → (name : String → String)
              → (f : ChoiceSet c₀ → ChoiceSet c₁) → {i : Size}
              → {lu : LUniv}
              → Process i {lu} c₀ → Process i {lu} c₁
fmapWithName name f (terminate a) = terminate (f a)
fmapWithName name f (node P)      = node (fmapWithName+ name f P)

```

```

fmapWithName+ : {c₀ c₁ : Choice}
              → (name : String → String)
              → (f : ChoiceSet c₀ → ChoiceSet c₁)
              → {i : Size}
              → {lu : LUniv}
              → Process+ i {lu} c₀ → Process+ i {lu} c₁
E (fmapWithName+ name f P) = E P
Lab (fmapWithName+ name f P) c = Lab P c
PE (fmapWithName+ name f P) c = fmapWithName∞ name f (PE P c)
I (fmapWithName+ name f P) = I P
PI (fmapWithName+ name f P) c = fmapWithName∞ name f (PI P c)
T (fmapWithName+ name f P) = T P
PT (fmapWithName+ name f P) c = f (PT P c)
Str+ (fmapWithName+ name f P) = name (Str+ P)

```

A.99 restrict.agda

```
--@PREFIX@Restrict
```

```
module restrict where
```

```

open import Data.Bool renaming (T to True)
open import Data.Sum
open import Data.Sum

```

```

open import Data.String renaming (_++_ to _++s_)
open import Data.String.Base
open import Size
open import process
open import labelUniv
open import showLabelP hiding (labelBoolFunToString)
open import choiceSetU
open import dataAuxFunction
open import auxData

-- restriction of external labels to those for which a function is true

-- ↑ is input as \uprightharpoonup

--@BEGIN@StricDef

_↑Str_ : {lu : LUniv} → String → (A : Label lu → Bool) → String
str ↑Str A = "Restrict " ++ labelBoolFunToString A ++ " " ++ str

mutual
  _↑∞_ : {lu : LUniv} {i : Size} → {c : Choice} → Process∞ i {lu} c
    → (A : Label lu → Bool) → Process∞ i {lu} c
  forcep (P ↑∞ A) = (forcep P) ↑ A
  Str∞ (P ↑∞ A) = (Str∞ P) ↑Str A

  _↑_ : {lu : LUniv} {i : Size} → {c : Choice} → Process i {lu} c
    → (A : Label lu → Bool) → Process i {lu} c
  terminate a ↑ A = terminate a
  node P ↑ A      = node (P ↑+ A)

  _↑+_ : {lu : LUniv} {i : Size} → {c : Choice} → Process+ i {lu} c
    → (A : Label lu → Bool) → Process+ i {lu} c
  E (P ↑+ A) = subset' (E P) (A ∘ (Lab P))
  Lab (P ↑+ A) (sub c p) = Lab P c
  PE (P ↑+ A) (sub c p) = PE P c ↑∞ A
  I (P ↑+ A)           = I P
  PI (P ↑+ A) c         = PI P c ↑∞ A
  T (P ↑+ A)           = T P

```

$$\begin{aligned} \text{PT } (P \mid+ A) \ c &= \text{PT } P \ c \\ \text{Str}+ (P \mid+ A) &= \text{Str}+ P \mid\text{Str} A \end{aligned}$$

--@END

A.100 sequentialComposition.agda

--@PREFIX@Sequential

module sequentialComposition where

open import Size
 open import Data.Sum
 open import choiceSetU
 open import process
 open import Data.String renaming (_++_ to _++s_)
 open import showFunction
 open import dataAuxFunction
 open import labelUniv

--@BEGIN@seq

$$\begin{aligned} _ \gg= \text{Str} _ &: \{c_0 : \text{Choice}\} \rightarrow \text{String} \\ &\rightarrow (\text{ChoiceSet } c_0 \rightarrow \text{String}) \rightarrow \text{String} \\ s \gg= \text{Str} f &= s ++s ";" ++s \text{choice2Str2Str } f \end{aligned}$$

mutual

$$\begin{aligned} _ \gg= \infty _ &: \{i : \text{Size}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ &\rightarrow \{lu : \text{LUniv}\} \\ &\rightarrow \text{Process} \infty \ i \ \{lu\} \ c_0 \\ &\rightarrow (\text{ChoiceSet } c_0 \rightarrow \text{Process} \infty \ i \ c_1) \\ &\rightarrow \text{Process} \infty \ i \ \{lu\} \ c_1 \\ \text{forcep } (P \gg= \infty \ Q) &= \text{forcep } P \gg= Q \\ \text{Str} \infty \ (P \gg= \infty \ Q) &= \text{Str} \infty \ P \gg= \text{Str} (\text{Str} \infty \circ Q) \end{aligned}$$

$$\begin{aligned} _ \gg= _ &: \{i : \text{Size}\} \rightarrow \{c_0 \ c_1 : \text{Choice}\} \\ &\rightarrow \{lu : \text{LUniv}\} \\ &\rightarrow \text{Process } i \ \{lu\} \ c_0 \end{aligned}$$

```

      → (ChoiceSet c0 → Process∞ (↑ i) c1)
      → Process i {lu} c1
node      P >>= Q = node (P >>=+ Q)
terminate x      >>= Q = forcep (Q x)

_>>=+_ : {i : Size} → {c0 c1 : Choice}
      → {lu : LUniv}
      → Process+ i {lu} c0
      → (ChoiceSet c0 → Process∞ i c1)
      → Process+ i {lu} c1
E (P >>=+ Q)      = E      P
Lab (P >>=+ Q)     = Lab    P
PE (P >>=+ Q) c    = PE P c >>=∞ Q
I (P >>=+ Q)       = I P Ψ' T P
PI (P >>=+ Q) (inj1 c) = PI P c >>=∞ Q
PI (P >>=+ Q) (inj2 c) = Q (PT P c)
T (P >>=+ Q)       = fin 0
PT (P >>=+ Q)      = ()
Str+ (P >>=+ Q) = Str+ P >>=Str (Str∞ ∘ Q)

--@END

_>>=+p_ : {i : Size} → {c0 c1 : Choice} → {lu : LUniv} → Process+ i {lu} c0
      → (ChoiceSet c0 → Process∞ i {lu} c1)
      → Process i {lu} c1
P >>=+p Q = node (P >>=+ Q)

```

A.101 sequentialCompositionRev.agda

module sequentialCompositionRev where

```

open import Size
open import Data.Sum
open import choiceSetU
open import process
open import Data.String      renaming (_++_ to _++s_)
open import showFunction
open import dataAuxFunction
open import labelUniv

```

```

_>>=Str_ : {c₀ : Choice} → String
           → (ChoiceSet c₀ → String) → String
s >>=Str f = s ++s ";" ++s choice2Str2Str f

```

mutual

```

_>>=∞_ : {i : Size} → {lu : LUniv}{c₀ c₁ : Choice}
         → Process∞ i {lu} c₀
         → (ChoiceSet c₀ → Process i {lu} c₁)
         → Process∞ i {lu} c₁
forcep (P >>=∞ Q)      = forcep P >>= Q
Str∞ (P >>=∞ Q) = Str∞ P >>=Str (Str ∘ Q)

```

```

_>>=_ : {i : Size} → {lu : LUniv}{c₀ c₁ : Choice}
        → Process i {lu} c₀
        → (ChoiceSet c₀ → Process i {lu} c₁)
        → Process i {lu} c₁

```

```

node      P >>= Q = node (P >>=+ Q)
terminate x >>= Q = Q x

```

```

_>>=+_ : {i : Size} → {lu : LUniv}{c₀ c₁ : Choice}
         → Process+ i {lu} c₀
         → (ChoiceSet c₀ → Process i {lu} c₁)
         → Process+ i {lu} c₁

```

```

E (P >>=+ Q)      = E P
Lab (P >>=+ Q)     = Lab P
PE (P >>=+ Q) c    = PE P c >>=∞ Q
I (P >>=+ Q)       = I P Ψ' T P
PI (P >>=+ Q) (inj₁ c) = PI P c >>=∞ Q
forcep (PI (P >>=+ Q) (inj₂ c)) = Q (PT P c)
Str∞ (PI (P >>=+ Q) (inj₂ c)) = Str (Q (PT P c))
T (P >>=+ Q)       = ∅'
PT (P >>=+ Q) ()   =
Str+ (P >>=+ Q) = Str+ P >>=Str (Str ∘ Q)

```

```

_>>=+p_ : {i : Size} → {lu : LUniv}{c₀ c₁ : Choice} → Process+ i {lu} c₀
         → (ChoiceSet c₀ → Process i {lu} c₁)
         → Process i {lu} c₁
P >>=+p Q = node (P >>=+ Q)

```

A.102 showFunction.agda

```

--@PREFIX@showFunction

module showFunction where

open import process
open import auxData
open import choiceSetU
open import Size
open import Data.List
open import Data.List.Base renaming (map to mapL)
open import Data.String renaming (_++_ to _++s_)
open import showLabelP hiding (showLabel)
open import choiceAuxFunction
open import Data.Bool renaming (T to True)
open import Data.Maybe
open import Data.Sum
open import labelUniv

--@BEGIN@showProLab

showProLab : { i : Size } → { c : Choice } → { lu : LUniv }
            → Process i {lu} c → String
showProLab (terminate x) = ""
showProLab {i}{c} (node P) = unlinesWithChosenString
    " "
    ((mapL (λ c' → extChoiceElToName (choice2Str {E} P} c')
        ++s " : "
        ++s showLabel (Lab P c'))
    (choice2Enum (E P)))
    ++
    (mapL (λ c → intChoiceElToName (choice2Str c)
        ++s " : "
        ++s "τ")
    (choice2Enum (I P))))

--@END

```

```

--@BEGIN@showtick

show✓ : { i : Size } → { c : Choice } → { lu : LUniv }
        → Process i {lu} c → String
show✓ (node P) = unlinesWithChosenString
                " "
                (mapL (λ t → (choice2Str t
                               ++s " : "
                               ++s choice2Str (PT P t)))
                      (choice2Enum (T P)))
show✓ (terminate a) = ""

--@END

--@BEGIN@proChoiceIempty

proChoicels∅ : { i : Size } → { c : Choice } → { lu : LUniv }
               → Process i {lu} c → Bool
proChoicels∅ (node P) = choicelsEmpty (E P) ∧ choicelsEmpty (I P)
proChoicels∅ (terminate x) = true

--@END

--@BEGIN@proHasSuccessfullyTerminated

proHasSuccessfullyTerminated : { i : Size } → { c : Choice } → { lu : LUniv }
                              → Process i {lu} c → Bool
proHasSuccessfullyTerminated (node P) = false
proHasSuccessfullyTerminated (terminate x) = true

--@END

--@BEGIN@proToI

proTol : ∀ { i } → { c : Choice } → { lu : LUniv } → Process i {lu} c → Choice
proTol (node P) = I P
proTol (terminate x) = fin 0

```

--@END

--@BEGIN@proToE

```
proToE : ∀ {i} → {c : Choice} → {lu : LUniv}
        → Process i {lu} c → Choice
proToE (node P) = E P
proToE (terminate x) = fin 0
```

--@END

--@BEGIN@proToLab

```
proToLab : ∀ {i} → {c : Choice} → {lu : LUniv}
        → (P : Process i {lu} c)
        → ChoiceSet (proToE P)
        → Label lu
proToLab (terminate x) ()
proToLab (node P) x = Lab P x
```

--@END

--@BEGIN@proPToSubProinf

```
proPToSubPro∞ : ∀ {i} → {c : Choice}
                → {lu : LUniv}
                → (P : Process i {lu} c)
                → ChoiceSet (proToE P) ⊔ ChoiceSet (proToL P)
                → Process∞ i {lu} c
proPToSubPro∞ (node P) (inj1 c') = PE P c'
proPToSubPro∞ (node P) (inj2 c') = PI P c'
proPToSubPro∞ (terminate x) (inj1 ())
proPToSubPro∞ (terminate x) (inj2 ())
```

--@END

```
--@BEGIN@proPToSubPrP
```

```
proPToSubPrP : ∀ {i} → {j : Size< i} → {c : Choice} → {lu : LUniv}
  → (P : Process i {lu} c)
  → ChoiceSet (proToE P) ⊔ ChoiceSet (proToI P)
  → Process j c
proPToSubPrP {i} {j} {c} P c' = forcep (proPToSubPro∞ {i} {c} P c')
```

```
--@END
```

```
--@BEGIN@choiceFunctionToString
```

```
choiceFunctionToString : {c₀ : Choice} → (c : Choice)
  → (g : ChoiceSet c → ChoiceSet c₀) → String
choiceFunctionToString {c₀} c g = unlinesWithChosenString
  " "
  (mapL (λ x → "(λ "
    ++s choice2Str x
    ++s " → "
    ++s choice2Str (g x)
    ++s ")")
    (choice2Enum c))
```

```
choiceFunctionToStringi : {c₀ : Choice} → {c : Choice}
  → (g : ChoiceSet c → ChoiceSet c₀) → String
choiceFunctionToStringi {c₀} {c} g = choiceFunctionToString {c₀} c g
```

```
--@END
```

```
--@BEGIN@choicetwoStrtwoStr
```

```
choice2Str2Str : {c : Choice} → (f : ChoiceSet c → String) → String
choice2Str2Str {c} f = unlinesWithChosenString " " (mapL ((λ x → "(λ "
  ++s (choice2Str x)
  ++s " → "
  ++s f x
  ++s ")")
  (choice2Enum c))
```

```
--@END
```

```
--@BEGIN@processToSubprocessz
```

```
processToSubprocess0 : ∀ {i} → {c : Choice} → {lu : LUniv}
  → (P : Process i {lu} c)
  → ChoiceSet (proToE P) ⊔ ChoiceSet (proToI P)
  → Process∞ i {lu} c
processToSubprocess0 (node P) (inj1 c') = PE P c'
processToSubprocess0 (node P) (inj2 c') = PI P c'
processToSubprocess0 (terminate x) (inj1 ())
processToSubprocess0 (terminate x) (inj2 ())
```

```
--@END
```

```
--@BEGIN@processToSubprocess
```

```
processToSubprocess : ∀ {i} → {j : Size< i} → {c : Choice} → {lu : LUniv}
  → (P : Process i {lu} c)
  → ChoiceSet (proToE P) ⊔ ChoiceSet (proToI P)
  → Process j c
processToSubprocess {i} {j} {c} P c' = forcep (processToSubprocess0 {i} {c} P c')
```

```
--@END
```

A.103 showFunctionForSimulator.agda

```
module showFunctionForSimulator where
```

```
open import process
open import auxData
open import choiceSetU
open import Size
```

```

open import Data.List
open import Data.List.Base renaming (map to mapL)
open import Data.String renaming (_++_ to _++s_)
open import showLabelP hiding (showLabel)
open import choiceAuxFunction
open import Data.Bool renaming (T to True)
open import Data.Maybe
open import Data.Sum
open import labelUniv

showMayLab : {lu : LUniv} → Maybe (Label lu) → String
showMayLab (just l) = showLabel l
showMayLab nothing = "τ"

showProLab : { i : Size} → {c : Choice} → {lu : LUniv} → Process i {lu} c → String
showProLab (terminate x) = ""
showProLab (node P) = unlinesWithChosenString
    " "
    ((mapL (λ c → extChoiceElToName (choice2Str c)
        ++s " : "
        ++s showLabel (Lab P c))
    (choice2Enum (E P)))
    ++
    (mapL (λ c → intChoiceElToName (choice2Str c)
        ++s " : "
        ++s "τ")
    (choice2Enum (I P))))

show✓ : { i : Size} → {c : Choice} → {lu : LUniv} → Process i {lu} c → String
show✓ (node P) = unlinesWithChosenString
    " "
    (mapL (λ t → (terminationChoiceElToName (choice2Str t)
        ++s " : "
        ++s choice2Str (PT P t))
    (choice2Enum (T P)))
show✓ (terminate a) = ""

```

```

proChoicels∅ : { i : Size } → { c : Choice } → { lu : LUniv } → Process i { lu } c → Bool
proChoicels∅ (node P) = choicelsEmpty (E P) ∧ choicelsEmpty (I P)
proChoicels∅ (terminate x) = true

```

```

proHasSuccessfullyTerminated : { i : Size } → { c : Choice } → { lu : LUniv } → Process i { lu } c → Bool
proHasSuccessfullyTerminated (node P) = false
proHasSuccessfullyTerminated (terminate x) = true

```

```

proToE : ∀ { i } → { c : Choice } → { lu : LUniv } → Process i { lu } c → Choice
proToE (node P) = E P
proToE (terminate x) = fin 0

```

```

proTol : ∀ { i } → { c : Choice } → { lu : LUniv } → Process i { lu } c → Choice
proTol (node P) = I P
proTol (terminate x) = fin 0

```

```

proPToSubPro∞ : ∀ { i } → { c : Choice }
  → { lu : LUniv }
  → (P : Process i { lu } c)
  → ChoiceSet (proToE P) ⊔ ChoiceSet (proTol P)
  → Process∞ i { lu } c
proPToSubPro∞ (node P) (inj1 c') = PE P c'
proPToSubPro∞ (node P) (inj2 c') = PI P c'
proPToSubPro∞ (terminate x) (inj1 ())
proPToSubPro∞ (terminate x) (inj2 ())

```

```

proPToSubPrP : ∀ { i } → { j : Size < i } → { c : Choice }
  → { lu : LUniv }
  → (P : Process i { lu } c)
  → ChoiceSet (proToE P) ⊔ ChoiceSet (proTol P)
  → Process j { lu } c
proPToSubPrP { i } { j } { c } P c' = forcep (proPToSubPro∞ { i } { c } P c')

```

```

choiceFunToStr : {c₀ : Choice} → (c : Choice)
               → (g : ChoiceSet c → ChoiceSet c₀) → String
choiceFunToStr {c₀} c g = unlinesWithChosenString
    " "
    (mapL (λ x → "(λ "
              ++s choice2Str x
              ++s " → "
              ++s choice2Str (g x)
              ++s ")"")
      (choice2Enum c))

```

```

choiceFunToStr↓ : {c₀ : Choice} → {c : Choice}
               → (g : ChoiceSet c → ChoiceSet c₀) → String
choiceFunToStr↓ {c₀} {c} g = choiceFunToStr {c₀} c g

```

```

choice2Str2Str : {c : Choice} → (f : ChoiceSet c → String) → String
choice2Str2Str {c} f = unlinesWithChosenString " " (mapL ((λ x → "(λ "
    ++s (choice2Str x)
    ++s " → "
    ++s f x
    ++s ")""))
    (choice2Enum c))

```

A.104 showLabelP.agda

```

module showLabelP where

```

```

open import label
open import Data.String renaming (_++_ to _++s_)
open import Data.String.Base
open import Data.Bool
open import Data.List.Base
open import Data.List

```

```

unlinesWithChosenString : String → List String → String
unlinesWithChosenString s [] = ""
unlinesWithChosenString s (s' :: []) = s'
unlinesWithChosenString s (s' :: s'' :: l) = s' ++s s ++s unlinesWithChosenString s (s'' :: l)

```

```

showLabel : Label → String
showLabel laba = "a"
showLabel labb = "b"
showLabel labc = "c"

```

```

LabelList : List Label
LabelList = laba :: labb :: labc :: []

```

```

labelBoolFunToString : (Label → Bool) → String
labelBoolFunToString f = unlines (map showLabel (filter f LabelList))

```

```

labelLabelFunToString : (Label → Label) → String
labelLabelFunToString f = "[["
    ++s unlinesWithChosenString ", " (map (λ l → showLabel (f l)
    ++s " <- " ++s showLabel l) LabelList )
    ++s "]]"

```

A.105 simulator.agda

```
--@PREFIX@main simulator
```

```
module simulator where
```

```

open import Size
open import Data.Bool
open import Data.Maybe
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import SizedIO.Base renaming (force to forceIO; delay to delayIO)
open import SizedIO.Console hiding (main)
open import choiceSetU
open import process
open import showFunction

```

```

open import choiceAuxFunction
open import Data.Sum
open import UnitModule
open import labelUniv

mutual

--@BEGIN@simulatorDefBlank

myProgram : ∀ {i} → (displayProcess : Bool) {lu : LUniv}
              (c₀ : Choice)
              → Process ∞ {lu} c₀
              → IOConsole i Unit
forceIO (myProgram {i} true c₀ P) =
  do' (putStrLn (Str P)) λ _ →
    myProgram₀ true c₀ P (proChoices∅ P)
    (proHasSuccessfullyTerminated P)
myProgram {i} false c₀ P =
  myProgram₀ false c₀ P (proChoices∅ P)
  (proHasSuccessfullyTerminated P)

--@END

--@BEGIN@simulatorDefPartZero

myProgram₀ : ∀ {i} → (displayProcess : Bool) (c₀ : Choice)
              {lu : LUniv} → Process ∞ {lu} c₀
              → (hasNoInternalOrExternalChoices : Bool)
              → (hasTerminated : Bool)
              → IOConsole i Unit
myProgram₀ displayProcess c₀ P false b =
  do (putStrLn
      ("Termination-Events: " ++ show✓ P)) λ _ →
  do (putStrLn
      ("Events: " ++ showProLab P)) λ _ →
  do (putStrLn ("Choose Event")) λ _ →
  myProgram₁ displayProcess c₀ P
myProgram₀ displayProcess c₀ P true false =
  do (putStrLn "Program got stuck") λ _ →
  return unit

```

```

myProgram0 displayProcess c0 P true true =
  do (putStrLn
      "Program has successfully terminated") λ _ →
    return unit

--@END

--@BEGIN@simulatorDefPartOne

myProgram1 : ∀ {i} → (displayProcess : Bool) → (c0 : Choice)
              → {lu : LUniv} → Process ∞ {lu} c0
              → IOConsole i Unit
forcelO (myProgram1 displayProcess c0 P) =
  do' getLine λ s →
    myProgram2 displayProcess c0 P s
    (s ==strb "quit")

--@END

--@BEGIN@simulatorDefPartTwo

myProgram2 : ∀ {i} → (displayProcess : Bool) (c0 : Choice)
              {lu : LUniv} (P : Process ∞ {lu} c0)
              → String → Bool
              → IOConsole i Unit
myProgram2 displayProcess c0 P s true =
  do (putStrLn "exiting") λ _ →
    return unit
myProgram2 displayProcess c0 P s false =
  myProgram3 displayProcess c0 P s (s ==strb "showProcess")

myProgram3 : ∀ {i} → (displayProcess : Bool) (c0 : Choice)
              {lu : LUniv} (P : Process ∞ {lu} c0)
              → String → Bool
              → IOConsole i Unit
forcelO (myProgram3 displayProcess c0 P s true) =
  do' (putStrLn (Str P)) λ _ →
    myProgram4 displayProcess c0 P
    (lookupChoice (proToE P) (proTol P) s)

```

```

myProgram3 displayProcess c0 P s false =
    myProgram4 displayProcess c0 P
    (lookupChoice (proToE P) (proTol P) s)

--@END

--@BEGIN@simulatorDefPartThree

myProgram4 : ∀ {i} → (displayProcess : Bool) (c0 : Choice) {lu : LUniv}
    (P : Process ∞ {lu} c0)
    → Maybe ((ChoiceSet (proToE P)) ⊔ (ChoiceSet (proTol P)))
    → IOConsole i Unit
forceIO (myProgram4 displayProcess c0 P nothing) =
    do' (putStrLn "please enter a choice amongst") λ _ →
    do (putStrLn (showProLab P)) λ _ →
    myProgram1 displayProcess c0 P
forceIO (myProgram4 displayProcess c0 P (just (inj1 ext))) =
    do' (putStrLn
        ("-" ++ showLabel (proToLab P ext) ++ s "→" ))
        λ _ →
    myProgram displayProcess c0 (proPToSubPrP P (inj1 ext))
forceIO (myProgram4 displayProcess c0 P (just (inj2 int))) =
    do' (putStrLn "-τ→") λ _ →
    myProgram displayProcess c0 (proPToSubPrP P (inj2 int))

--@END

--@BEGIN@simulatorDefParti

myProgrami : ∀ {i} → (displayProcess : Bool){c0 : Choice}{lu : LUniv}
    → Process ∞ {lu} c0 → IOConsole i Unit
myProgrami {i} displayProcess {c0} P = myProgram {i} displayProcess c0 P

--@END

myProgrami∞ : (displayProcess : Bool) {c0 : Choice} {lu : LUniv}
    → Process∞ ∞ {lu} c0 → IOConsole ∞ Unit
myProgrami∞ displayProcess P = myProgrami {∞} displayProcess (forceP P)

```

```
compile : {c : Choice}{lu : LUniv}(p : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  NativeIO Unit
compile p = translateIOConsole (myProgrami $\infty$  true p)
```

A.106 simulatorCutDown.agda

```
module simulatorCutDown where
```

```
open import Size
open import Data.Sum
open import Data.Bool
open import Data.Maybe
open import Data.List
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import SizedIO.Base renaming (force to forceIO; delay to delayIO)
open import SizedIO.Console hiding (main)
open import NativeIO
open import choiceSetU
open import process
open Process $\infty$ 
open Process+
open import showFunction
open import choiceAuxFunction
open import label
open import prefix
open import primitiveProcess
open import interleave
open import UnitModule
open import labelUniv
```

```
mutual
```

```
simulator :  $\forall$  {i}  $\rightarrow$  {lu : LUniv}{c0 : Choice}
 $\rightarrow$  Process  $\infty$  {lu} c0  $\rightarrow$  IOConsole i Unit
forceIO (simulator P) =
  do' (putStrLn (Str P))  $\lambda$  _  $\rightarrow$ 
  do (putStrLn ("Termination-Events:"
    ++s show $\checkmark$  P))  $\lambda$  _  $\rightarrow$ 
```

```

do (putStrLn
    ("Events" ++s showProLab P)) λ _ →
do (putStrLn ("Choose Event"))      λ _ →
do getLine                          λ s →
simulator1 P (lookupChoice
    (proToE P)
    (proTol P) s)

simulator1 : ∀ {i} → {lu : LUniv}{c0 : Choice}
    → (P : Process ∞ {lu} c0)
    → Maybe ((ChoiceSet (proToE P))
        ⊔ (ChoiceSet (proTol P)))
    → IOConsole i Unit
forcelO (simulator1 P nothing) =
do' (putStrLn
    "please enter a choice amongst") λ _ →
do (putStrLn (showProLab P))      λ _ →
simulator P
simulator1 P (just c1) =
simulator (proPToSubPrP P c1)

setSTOP : Choice
setSTOP = namedElements ("STOP" :: [])

transition1 : ∀ i → Process i {lsimple} setSTOP
transition1 i = (lab laba) → delay ((lab labb) → delay ((lab labc) → delay (STOP {∞} setSTOP

transition2 : ∀ i → Process i {lsimple} setSTOP
transition2 i = (lab laba) → delay ((lab labb) → delay ((lab labc) → delay {↑ (↑ i)} (STOP {∞}

myResultType : Choice
myResultType = setSTOP ×' setSTOP

myProcess : Process ∞ {lsimple} myResultType
myProcess = transition1 ∞ ||| transition2 ∞

main : NativeIO Unit
main = translatoConsole (simulator myProcess)

```

A.107 skip.agda

module skip where

```
open import Size
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List
open import process
open import auxData
open import dataAuxFunction
open import choiceSetU
open import labelUniv
```

```
STOP+ : (i : Size) → {lu : LUniv} → (c : Choice) → Process+ i {lu} c
STOP+ i c = process+ ∅' efq efq ∅' efq ∅' efq "STOP"
```

```
STOP : (i : Size) → {lu : LUniv} → (c : Choice) → Process i {lu} c
STOP i c = node (process+ ∅' efq efq ∅' efq ∅' efq "STOP")
```

```
SKIP+ : {lu : LUniv} {i : Size} → {c : Choice} → (a : ChoiceSet c)
      → Process+ i {lu} c
SKIP+ {lu} a = process+ ∅' efq efq ∅' efq ⊤' (λ _ → a)
      ("SKIP(" ++s choice2Str a ++s ")")
```

```
SKIP : {lu : LUniv} {i : Size} → {c : Choice} → (a : ChoiceSet c)
      → Process i {lu} c
SKIP {lu} a = node (SKIP+ a)
```

```
TERMINATE : {i : Size} → {lu : LUniv} {c : Choice} → (a : ChoiceSet c) → Process i {lu} c
TERMINATE a = terminate a
```

```
TERMINATE∞ : {i : Size} → {lu : LUniv} {c : Choice} → (a : ChoiceSet c) → Process∞ i {lu} c
forcep (TERMINATE∞ a) = TERMINATE a
Str∞ (TERMINATE∞ a) = "terminate(" ++s choice2Str a ++s ")"
```

```
SKIPL+ : {lu : LUniv}{i : Size} → {c : Choice} → List (ChoiceSet c) → Process+ i {lu} c
SKIPL+ {lu} l = process+ ∅' efq efq ∅' efq (fin (length l)) (nth l) "SKIPL"
```

A.108 test.agda

```
--@PREFIX@main
module test where

--@BEGIN@natDef
data N : Set where
  zero : N
--@HIDE-BEG
  suc : N → N
--@HIDE-END
--@END

--@BEGIN@colDef
data color : Set where
  Red : color
  Green : color
--@HIDE-BEG
  Blue : color
--@HIDE-END
--@END
```

```
open import Data.Bool
```

A.109 theoremProverAgdaChapterCod.agda

```
--@PREFIX@main
module theoremProverAgdaChapterCod where

--@BEGIN@colDef
```



```
data color : Set where
  Red : color
  Green : color
  Blue : color
```

```
--@END
```

```
--@BEGIN@swapCol
```

```
swapColor : color → color
swapColor Red = Green
swapColor Green = Red
```

```
--@END
```

```
--@HIDE-BEG
swapColor Blue = Red
--@HIDE-END
```

```
--@BEGIN@natDef
```

```
data N : Set where
  zero : N
  succ : N → N
```

```
--@END
```

```
--@BEGIN@plusOp
```

```
_+_ : N → N → N
zero   + m = m
succ n + m = succ (n + m)
```

```
--@END
```

```
--@BEGIN@doublOp
```

```
double : N → N
```



```

double zero = zero
double (succ n) = succ (double n)

--@END

--@BEGIN@fibDef

fib : ℕ → ℕ
fib zero = zero
fib (succ zero) = succ zero
fib (succ (succ n)) = fib n + fib (succ n)

--@END

open import Coinduction using (#_ ; ♭ ; ∞)
open import Size renaming (∞ to ∞')

--@BEGIN@streamDef

data stream : Set where
  _::_ : ℕ → ∞ stream → stream

--@END

--@BEGIN@streamZero

stream₀ : stream
stream₀ = zero :: (# stream₀)

--@END

--@BEGIN@contStream

inc : ℕ → stream
inc n = n :: (# inc (succ n))

--@END

--@BEGIN@comStream

```

```

addStream : stream → stream → stream
addStream (x :: x₁) (y :: y₁) = (x + y) :: (⊕ addStream (⌞ x₁) (⌞ y₁))

--@END

--@BEGIN@deDefination

mutual
  record ∞Delay (i : Size) (A : Set) : Set where
    coinductive
    field
      force : {j : Size < i} → Delay j A

  data Delay (i : Size) (A : Set) : Set where
    now : A → Delay i A
    later : ∞Delay i A → Delay i A

--@END

--@BEGIN@BoolDef

data Bool : Set where
  true : Bool
  false : Bool

--@END

--@BEGIN@orDef

_or_ : Bool → Bool → Bool
false  or  m =  m
true   or  m = true

--@END

```

```
--@BEGIN@postulateDef

postulate _and_ : Bool → Bool → Bool
postulate if_then_else_ : Bool → Bool → Bool → Bool

--@END

--@BEGIN@infixlDef

infixl    10 _or_
infixl    11 _and_
infixl    12 if_then_else_

--@END

--@BEGIN@boolDef

data B : Set where
  true  : B
  false : B

--@END

--@BEGIN@orUnicode

_∨_ : B → B → B
true  ∨ true  = true
true  ∨ false = true
false ∨ true  = true
false ∨ false = false

--@END

--@BEGIN@idExEx

id : {A : Set} → A → A
id x = x

--@END

--@BEGIN@idImEx
```



```
id1 : (A : Set) → A → A
id1 A x = x
```

```
--@END
```

```
--@BEGIN@idboolExEx
```

```
True : Bool
True = id true
```

```
--@END
```

```
--@BEGIN@idzeroExEx
```

```
Zero : ℕ
Zero = id zero
```

```
--@END
```

```
--@BEGIN@idNatImEx
```

```
Zero1 : ℕ
Zero1 = id1 ℕ zero
```

```
--@END
```

```
--@BEGIN@idBoolImEX
```

```
true1 : Bool
true1 = id1 Bool true
```

```
--@END
```

```
--@BEGIN@importMayEx
```

```
import maybe
--@END
```

```
--@BEGIN@fMayBeEx
```



```
fMaybe : {A B : Set} → (A → B) → maybe.Maybe A → maybe.Maybe B
fMaybe f maybe.nothing = maybe.nothing
fMaybe f (maybe.just x) = maybe.just (f x)
```

```
--@END
```

```
--import Data.Nat renaming ( N to N' )
--open Data.Nat
```

```
--@BEGIN@zeroExNat
```

```
Z : N
Z = zero
```

```
--@END
```

```
--@BEGIN@openImportEx
```

```
open import maybe
--@END
```

```
--@BEGIN@fmaybeOpenImportEx
```

```
fMaybe1 : {A B : Set} → (A → B) → Maybe A → Maybe B
fMaybe1 f nothing = nothing
fMaybe1 f (just x) = just (f x)
```

```
--@END
```

```
--@BEGIN@recordDef
```

```
record AB : Set where
  constructor pair
  field
    a : N
```



```

    b : Bool

--@END
open AB

postulate x : ℕ
postulate y : Bool

--@BEGIN@recordDefOne

n1 : AB
n1 = pair x y

--@END

n2 : AB

--@BEGIN@recordDefTwo

n2 = record{a = x; b = y}

--@END

n3 : AB
--@BEGIN@recordDefThree

a  n3 = x
b  n3 = y

--@END

--@BEGIN@postulateExample

postulate A : Set
postulate a' : A
postulate _==_ : A → A → Set
postulate _>'_ : A → A → Set

```



```

--@END

--@BEGIN@listDef

data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

--@END

--@BEGIN@whereEx

revList : (A : Set) → List A → List A
revList A list = refAux list []
  where
    refAux : List A → List A → List A
    refAux [] xy = xy
    refAux (x :: xs) xy = refAux xs (x :: xy)

--@END

--@BEGIN@mutualEx

mutual
  data Even : Set where
    zero : Even
    suc : Odd → Even

  data Odd : Set where
    suc : Even → Odd

--@END

--@BEGIN@mutualExtwo

data Even' : Set
data Odd' : Set

```

```

data Even' where
  zero : Even'
  suc  : Odd' → Even'

data Odd' where
  suc : Even' → Odd'

--@END

_<_ : ℕ → ℕ → Bool
_ < zero = false
zero < succ m = true
succ n < succ m = n < m

--@BEGIN@withEx

MinN : ℕ → ℕ → ℕ
MinN x y with (x < y)
MinN x y | true = x
MinN x y | false = y

--@END

--open import Data.Nat

data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

--{-# BUILTIN NATURAL Nat #-}

--{-# BUILTIN NATURAL ℕ #-}

```

```
{- aa1 :  ℕ aa1 = 0 -}
```

```
{- {-# BUILTIN LIST List #-} {-# BUILTIN NIL [] #-} {-# BUILTIN CONS _::_ #-} -}
```

```
data List0 (A : Set) : Set where
  [] : List0 A
  _::_ : A → List0 A → List0 A
```

```
--@END
```

```
--{-# COMPILED_DATA List List [] (::) #-}
```

```
--@BEGIN@listPragma
```

```
{-# FOREIGN GHC type AgdaList a = [a] #-}
{-# COMPILE GHC List = data AgdaList ([] | (::)) #-}
```

```
--@END
```

```
--@BEGIN@IOPostCom
```

```
postulate IO : Set → Set
--@END
```

```
--{-# COMPILED_TYPE IO IO #-}
-- {-# FOREIGN GHC type AgdaIO a = IO a #-}
-- {-# COMPILE GHC IO = type AgdaIO #-}
```

```
--@BEGIN@CoIO
```

```
{-# COMPILE GHC IO = type IO #-}
```

```
--@END
```

```
--@BEGIN@UnDef
```

```

data Unit : Set where
  unit : Unit

--@END

--{-# COMPILED_DATA Unit () () #-}

--@BEGIN@coIOEx

{-# COMPILE GHC Unit = data () (() #-}

--@END

--@BEGIN@postulateStringEx

postulate String : Set
{-# BUILTIN STRING String #-}

postulate putStrLn : String → IO Unit
{-# COMPILE GHC putStrLn = (\ s -> putStrLn (Data.Text.unpack s)) #-}

--@END
-- {-# COMPILED putStrLn (\ _ s -> Data.Text.IO.putStrLn s) #-}
--@BEGIN@comStrEx

--{-# COMPILED_TYPE String String #-}

--{-# COMPILED putStrLn putStrLn #-}

--@END

--@BEGIN@botEx

data ⊥ : Set where

--@END

--@BEGIN@topEx

data ⊤ : Set where
  triv : ⊤

```

```

--@END

--@BEGIN@inductivDef

mutual
  data U : Set where
    ⊥'      : U
    ⊤'      : U
    Bool'   : U
    Π'      : (a : U)(b : T0 a → U) → U

  T0 : U → Set
  T0 ⊥'      = ⊥
  T0 ⊤'      = ⊤
  T0 Bool'   = Bool
  T0 (Π' a b) = (x : T0 a) → T0 (b x)

```

```

--@END

```

```

--@BEGIN@StreamDef

record Stream (i : Size) : Set where
  coinductive
  field
    head : ℕ
    tail  : {j : Size < i} → Stream j

```

```

--@END

```

```

open Stream

```

```
--@BEGIN@consAgdaCode
```

```
cons : ∀ {i} → ℕ → Stream i → Stream (↑ i)
head (cons n s)    = n
tail (cons n s)    = s
```

```
--@END
```

```
--@BEGIN@plusStreamAgdaCode
```

```
_+_ : ∀ {i} → Stream i → Stream i → Stream i
head (s   +_ s') = head s +  head s'
tail (s   +_ s') = tail s +_ tail s'
```

```
--@END
```

```
--@BEGIN@isTrue
```

```
T : Bool → Set
T true =  ⊤
T false = ⊥
```

```
--@END
```

```
infix 4 _≡_
```

```
-- \maine
--@BEGIN@DefEq
```

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

```
--@END
```

```
{-# BUILTIN EQUALITY _≡_ #-}
```

```
--@BEGIN@rewriteEx
```

```
+0 : ∀ n → n + zero ≡ n
+0 zero = refl
+0 (succ n) rewrite +0 n = refl
```

```
--@END
```

```
--@BEGIN@rewriteExDecoded
```

```
+0' : ∀ n → n + zero ≡ n
+0' zero = refl
+0' (succ n) with n + zero | +0' n
+0' (succ n) | .n | refl = refl
```

```
--@END
```

```
-- \maine
--@BEGIN@Fin
```

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (succ n)
  suc  : {n : ℕ} → Fin n → Fin (succ n)
```

```
--@END
```

A.110 theoremProverAgdaChapterCod2.agda

```
--@PREFIX@maineTwo
module theoremProverAgdaChapterCod2 where
```

```
--@BEGIN@colDef
```

```
-- verison of theoremProverAgdaChapterCod
-- where when two variants are the variants have no special symbol

data color : Set where
  Red : color
  Green : color
  Blue : color

--@END

--@BEGIN@swapCol

swapColor : color → color
swapColor Red = Green
swapColor Green = Red

--@END

--@HIDE-BEG
swapColor Blue = Red
--@HIDE-END

--@BEGIN@natDef

data N : Set where
  zero : N
  succ : N → N

{-# BUILTIN NATURAL N #-}

--@END

--@BEGIN@plusOp

_+_ : N → N → N
zero   + m = m
succ n  + m = succ (n + m)
```

```
--@END

--@BEGIN@doubleOp

double : ℕ → ℕ
double zero = zero
double (succ n) = succ (double n)

--@END

--@BEGIN@fibDef

fib : ℕ → ℕ
fib zero = zero
fib (succ zero) = succ zero
fib (succ (succ n)) = fib n + fib (succ n)

--@END

open import Coinduction using (#_ ; ♭ ; ∞)
open import Size renaming (∞ to ∞')

--@BEGIN@streamDef

data stream : Set where
  _::_ : ℕ → ∞ stream → stream

--@END

--@BEGIN@streamZero

strem0 : stream
strem0 = zero :: (# strem0)

--@END

--@BEGIN@contStream

inc : ℕ → stream
```

```

inc n = n :: (# inc (succ n))

--@END

--@BEGIN@comStream

addStream : stream → stream → stream
addStream (x :: x₁) (y :: y₁) = (x + y) :: (# addStream (b x₁) (b y₁))

--@END

--@BEGIN@deDefination

mutual
  record ∞Delay (i : Size) (A : Set) : Set where
    coinductive
    field
      force : {j : Size < i} → Delay j A

  data Delay (i : Size) (A : Set) : Set where
    now : A → Delay i A
    later : ∞Delay i A → Delay i A

--@END


data Bool : Set where
  true : Bool
  false : Bool


--@BEGIN@orDef

_or_ : Bool → Bool → Bool
false or m = m
true or m = true

```

```
--@END

--@BEGIN@postulateDef

postulate _and_ : Bool → Bool → Bool
postulate if_then_else_ : Bool → Bool → Bool → Bool

--@END

--@BEGIN@infixlDef

infixl 10 _or_
infixl 11 _and_
infixl 12 if_then_else_

--@END

--@BEGIN@boolDef

data  $\mathbb{B}$  : Set where
  true  :  $\mathbb{B}$ 
  false :  $\mathbb{B}$ 

--@END

--@BEGIN@orUnicode

_∨_ :  $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$ 
true  ∨ true  = true
true  ∨ false = true
false ∨ true  = true
false ∨ false = false

--@END

--@BEGIN@idExEx

id : {A : Set} → A → A
id x = x
```

--@END

--@BEGIN@idImEx

$\text{id}_1 : (A : \text{Set}) \rightarrow A \rightarrow A$
 $\text{id}_1 \ A \ x = x$

--@END

--@BEGIN@idboolExEx

$\text{True} : \text{Bool}$
 $\text{True} = \text{id} \ \text{true}$

--@END

--@BEGIN@idzeroExEx

$\text{Zero} : \mathbb{N}$
 $\text{Zero} = \text{id} \ \text{zero}$

--@END

--@BEGIN@idNatImEx

$\text{Zero}_1 : \mathbb{N}$
 $\text{Zero}_1 = \text{id}_1 \ \mathbb{N} \ \text{zero}$

--@END

--@BEGIN@idBoolImEX

$\text{true}_1 : \text{Bool}$
 $\text{true}_1 = \text{id}_1 \ \text{Bool} \ \text{true}$

--@END

--@BEGIN@importMayEx

import maybe

```
--@END

--import Data.Nat renaming ( N to N')
--open Data.Nat

--@BEGIN@zeroExNat

Z : ℕ
Z = zero

--@END

--@BEGIN@openImportEx

open import maybe
--@END

--@BEGIN@fmaybeOpenImportEx

fMaybe : {A B : Set} → (A → B) → Maybe A → Maybe B
fMaybe f nothing = nothing
fMaybe f (just x) = just (f x)

--@END

--@BEGIN@recordDef

record AB : Set where
  constructor pair
  field
    a : ℕ
    b : Bool

--@END
open AB
```



```
postulate x : ℕ
postulate y : Bool

--@BEGIN@recordDefOne

n1 : AB
n1 = pair x y

--@END

n2 : AB

--@BEGIN@recordDefTwo

n2 = record{a = x; b = y}

--@END

n3 : AB
--@BEGIN@recordDefThree

a  n3 = x
b  n3 = y

--@END

--@BEGIN@postulateExample

postulate A : Set
postulate a' : A
postulate ==_ : A → A → Set
postulate >'_ : A → A → Set

--@END

--@BEGIN@List
```



```

data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

--@END

{-# BUILTIN LIST List #-}

--@BEGIN@whereEx

revList : (A : Set) → List A → List A
revList A list = refAux list []
  where
    refAux : List A → List A → List A
    refAux []      xy = xy
    refAux (x :: xs) xy = refAux xs (x :: xy)

--@END

--@BEGIN@mutualExtwo

data Even : Set
data Odd  : Set

data Even where
  zero : Even
  suc  : Odd → Even

data Odd where
  suc : Even → Odd

--@END

_<_ : ℕ → ℕ → Bool
_ < zero = false
zero < succ m = true
succ n < succ m = n < m

```

```
--@BEGIN@withEx
```

```
MinN1 : ℕ → ℕ → ℕ
MinN1 x y with (x < y)
MinN1 x y | true = x
MinN1 x y | false = y
```

```
--@END
```

```
--@BEGIN@withminEx
```

```
MinN : ℕ → ℕ → ℕ
MinN x y with (x < y)
... | true = x
... | false = y
```

```
--@END
```

```
--@BEGIN@nestedPatt
```

```
mutual
  f : ℕ → ℕ
  f zero           = 1
  f (succ zero)    = 2
  f (succ (succ x)) = g x

  g : ℕ → ℕ
  g zero = 3
  g (succ n) = n
```

```
--@END
```

```
--open import Data.Nat
```

```

data Nat : Set where
  zero : Nat
  suc   : Nat → Nat

--{-# BUILTIN NATURAL Nat #-}

--{-# BUILTIN NATURAL ℕ #-}

{- aa1 : ℕ aa1 = 0 -}

{- {-# BUILTIN LIST List #-} {-# BUILTIN NIL [] #-} {-# BUILTIN CONS _::_ #-} -}

--@BEGIN@listDef

data List0 (A : Set) : Set where
  [] : List0 A
  _::_ : A → List0 A → List0 A

--@END

--@BEGIN@listPragma

--{-# COMPILED_DATA List List [] (::) #-}
{-# COMPILE GHC List = data MAlonzo.Code.Agda.Builtin.List.AgdaList ([] | (::)) #-}

--@END

--@BEGIN@IOPostCom

postulate IO : Set → Set

--@END

--@BEGIN@CoIO

```

```

--{-# COMPILED_TYPE IO IO #-}
{-# COMPILE GHC IO = type MAlonzo.Code.Agda.Builtin.IO.AgdaIO #-}

--@END

--@BEGIN@UnDef

data Unit : Set where
  unit : Unit

--@END

--@BEGIN@coIOEx

--{-# COMPILED_DATA Unit () () #-}
{-# COMPILE GHC Unit = data () (()) #-}

--@END

--@BEGIN@postulateStringEx

postulate String : Set
postulate putStrLn : String → IO Unit

--@END

--@BEGIN@comStrEx

--{-# COMPILED_TYPE String String #-}

--{-# COMPILED putStrLn putStrLn #-}

--@END

--@BEGIN@botEx

data ⊥ : Set where

--@END

--@BEGIN@topEx

```

```

data T : Set where
  triv : T

--@END

--@BEGIN@inductivDef

mutual
  data U : Set where
    ⊥'   : U
    ⊤'   : U
    Bool' : U
    Π'   : (a : U)(b : T0 a → U) → U

  T0 : U → Set
  T0 ⊥'   = ⊥
  T0 ⊤'   = ⊤
  T0 Bool' = Bool
  T0 (Π' a b) = (x : T0 a) → T0 (b x)

--@END

--@BEGIN@StreamDef

record Stream (i : Size) : Set where
  coinductive
  field
    head : ℕ
    tail : {j : Size < i} → Stream j

--@END

```

```
open Stream
```

```
--@BEGIN@consAgdaCode
```

```
cons : ∀ {i} → ℕ → Stream i → Stream (↑ i)
head (cons n s) = n
tail (cons n s) = s
```

```
--@END
```

```
--@BEGIN@plusStreamAgdaCode
```

```
_+_ : ∀ {i} → Stream i → Stream i → Stream i
head (s   +s s') = head s   + head s'
tail (s   +s s') = tail s   +s tail s'
```

```
--@END
```

```
-- \mainTwo
```

```
--@BEGIN@IsEven
```

```
data IsEven : ℕ → Set where
  even0      : IsEven 0
  evenSuc    : {n : ℕ} → IsEven n → IsEven (succ (succ n))
```

```
--@END
```

```
-- \mainTwo
```

```
--@BEGIN@evenplus
```

```

even+ : (n m : ℕ) → IsEven n → IsEven m → IsEven (n + m)
even+ .0 m even0 pm = pm
even+ .(succ (succ n)) m (evenSuc {n} pn) pm = evenSuc (even+ n m pn pm)

--@END

```

A.111 theoremProverAgdaChapterCod3.agda

```

--@PREFIX@mainethree
module theoremProverAgdaChapterCod3 where

open import Size
open import Data.String
open import Data.Unit

-- \mainethree
--@BEGIN@Maybe

data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A

--@END

-- \mainethree
--@BEGIN@IOInterface

record IOInterface : Set1 where
  field
    Command : Set
    Response : Command → Set

--@END
open IOInterface public

mutual
-- \mainethree

```

```

--@BEGIN@IO

record IO (i : Size) (I : IOInterface) (A : Set) : Set where
  coinductive
  constructor delay
  field
    force : {j : Size < i} → IO' j I A

data IO' (i : Size) (I : IOInterface) (A : Set) : Set where
  do' : (c : I.Command) (f : I.Response c → IO i I A) → IO' i I A
  return' : (a : A) → IO' i I A

--@END

data IO+ (i : Size) (I : IOInterface) (A : Set) : Set where
  do' : (c : I.Command) (f : I.Response c → IO i I A) → IO+ i I A

open IO public

-- \mainethree
--@BEGIN@ConsoleCommand

data ConsoleCommand : Set where
  putStrLn : String → ConsoleCommand
  getLine : ConsoleCommand

ConsoleResponse : ConsoleCommand → Set
ConsoleResponse (putStrLn s) = ⊤
ConsoleResponse getLine     = String

consolel : IOInterface
consolel .Command = ConsoleCommand
consolel .Response = ConsoleResponse

IOConsole : Size → Set → Set
IOConsole i = IO i consolel

--@END

module _ {I : IOInterface} where

```

```

infixl 2 _>=>_ _>=>' _>=>_

mutual
-- \mainethree
--@BEGIN@monadicBind

_>=>'_ : ∀{i}{A B : Set}(m : IO' i I A) (k : A → IO (↑ i) I B) → IO' i I B
do' c f >=>' k = do' c λ x → f x >=> k
return' a >=>' k = (k a) .force

_>=>_ : ∀{i}{A B : Set}(m : IO i I A) (k : A → IO i I B) → IO i I B
(m >=> k) .force = m .force >=>' k

_>_ : ∀{i}{B : Set} (m : IO i I ⊤) (k : IO i I B) → IO i I B
m > k = m >=> λ _ → k

--@END

```

A.112 traceEquivalence.agda

```

--@PREFIX@maintraceEquivalence

module traceEquivalence where

open import Size
open import Data.List
open import Data.Product
open import Data.Maybe
open import label
open import process
open import choiceSetU
open import TraceWithoutSize
open import RefWithoutSize
open import labelUniv

--@BEGIN@TrEqDef

```

```


$$\_ \equiv \_ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} \infty \{lu\} \ c_0) \rightarrow \text{Set}$$


$$P \equiv Q = P \sqsubseteq Q \times Q \sqsubseteq P$$


--@END

--@BEGIN@TrEqInfDef


$$\_ \equiv \infty \_ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} \infty \infty \{lu\} \ c_0) \rightarrow \text{Set}$$


$$P \equiv \infty Q = P \sqsubseteq \infty Q \times Q \sqsubseteq \infty P$$


--@END

--@BEGIN@TrEqPlusDef


$$\_ \equiv + \_ : \{lu : \text{LUniv}\} \{c_0 : \text{Choice}\} \rightarrow (P \ Q : \text{Process} + \infty \{lu\} \ c_0) \rightarrow \text{Set}$$


$$P \equiv + Q = P \sqsubseteq + Q \times Q \sqsubseteq + P$$


--@END

```

A.113 traceImpliesTraceP.agda

```

module traceImpliesTraceP where

open import process
open import choiceSetU
open import labelUniv
open import Size
open import Data.Empty
open import Data.List
open import Data.Maybe
open import Data.Sum
open import TraceWithNextProcess
open import TraceWithoutSize
open import bisimilarity

```

mutual

```

termEquivalentImpliesTrace : {lu : LUniv}{c : Choice}{x : ChoiceSet c}
  (P : Process ∞ {lu} c)(terequiv : TerminateEquivalent x P)
  → Tr [] (just x) P
termEquivalentImpliesTrace {lu}{c} {x} .(terminate x) termeqterm = ter x
termEquivalentImpliesTrace {lu}{c} {x} .(node Q) (termeqnode {Q} terequiv) =
  termEquivalentImpliesTraceaux Q terequiv (hasTauOrTick terequiv)

```

```

termEquivalentImpliesTraceaux : {lu : LUniv}{c : Choice}{x : ChoiceSet c}
  (Q : Process+ ∞ {lu} c)(terequiv : TerminateEquivalent+ x Q)
  → (hasTauOrTick : ChoiceSet (I Q) ⊔ ChoiceSet (T Q) )
  → Tr [] (just x) (node Q)
termEquivalentImpliesTraceaux {lu}{c} {x} Q terequiv (inj1 ic) = tnode (intc [] (just x) ic
  (termEquivalentImpliesTrace∞ (PI Q ic) (onlyIntChoice
termEquivalentImpliesTraceaux {lu}{c} {x} Q terequiv (inj2 tc) rewrite (termIsa terequiv tc) = tnode

```

```

termEquivalentImpliesTrace∞ : {lu : LUniv}{c : Choice}{x : ChoiceSet c}(P : Process∞ ∞ {lu} c)
  (terequiv : TerminateEquivalent∞ x P)
  → Tr∞ [] (just x) P
termEquivalentImpliesTrace∞ {lu}{c} {x} P terequiv = termEquivalentImpliesTrace (forcep P) terequiv

```

mutual

```

termEquivalentImpliesTraceEmpty : {lu : LUniv}{c : Choice}{x : ChoiceSet c}(P : Process ∞ {lu} c)
  (terequiv : TerminateEquivalent x P) → Tr [] nothing P
termEquivalentImpliesTraceEmpty {lu} {c} {x} .(terminate x) termeqterm = empty x
termEquivalentImpliesTraceEmpty {lu} {c} {x} .(node Q) (termeqnode {Q} terequiv) =
  termEquivalentImpliesTraceEmptyaux Q terequiv (hasTauOrTick terequiv)

```

```

termEquivalentImpliesTraceEmptyaux : {lu : LUniv}{c : Choice}{x : ChoiceSet c}(Q : Process+ ∞ {lu} c)
  (terequiv : TerminateEquivalent+ x Q) → Tr [] nothing Q

```

```

      (terequiv : TerminateEquivalent+ x Q)
    → (hasTauOrTick : ChoiceSet (I Q) ⊔ ChoiceSet (T Q) )
    → Tr [] nothing (node Q)
termEquivalentImpliesTraceEmptyaux {lu}{c} {x} Q terequiv (inj1 ic) = tnode (intc [] nothing)
      (termEquivalentImpliesTraceEmpty∞ (PI Q ic) (onlyIntChoiceSet c))
termEquivalentImpliesTraceEmptyaux {lu}{c} {x} Q terequiv (inj2 tc) rewrite (termIsa terequiv)

```

```

termEquivalentImpliesTraceEmpty∞ : {lu : LUniv}{c : Choice}{x : ChoiceSet c}(P : Process+
      (terequiv : TerminateEquivalent∞ x P)
    → Tr∞ [] nothing P
termEquivalentImpliesTraceEmpty∞ {lu}{c} {x} P terequiv = termEquivalentImpliesTraceEmpty∞

```

mutual

```

termEquivalentTracelsTerTrace+ : {lu : LUniv}{c : Choice}{x : Maybe (ChoiceSet c)}
      {a : ChoiceSet c}(P : Process+ ∞ {lu} c)(l : List (Label lu))
      (terequivP : TerminateEquivalent a (node P))(tr : Tr+ l x)
    → Tr l x (terminate a)
termEquivalentTracelsTerTrace+ {lu}{c} {.nothing} {a} P .[] (termeqnode x1) empty = empty
termEquivalentTracelsTerTrace+ {lu}{c} P .(Lab P x :: l) (termeqnode terequivP) (extc l tick) =
termEquivalentTracelsTerTrace+ {lu}{c} P l (termeqnode terequivP) (intc .l tick x tr) =
      termEquivalentTracelsTerTrace∞ (PI P x) l (onlyIntChoiceSet c)
termEquivalentTracelsTerTrace+ {lu}{c} {.just (PT P x))} {a} P .[]
      (termeqnode terequivP) (terc x) rewrite termIsa terequivP

```

```

termEquivalentTracelsTerTrace∞ : {lu : LUniv}{c : Choice}{x : Maybe (ChoiceSet c)}{a : ChoiceSet c}
      (P : Process∞ ∞ {lu} c)(l : List (Label lu))
      (terequivP : TerminateEquivalent∞ a P)(tr : Tr∞ l x P)
    → Tr l x (terminate a)
termEquivalentTracelsTerTrace∞ {lu}{c} {x} {a} P l terequivP tr =
      termEquivalentTracelsTerTrace (forceP P)

```

```

termEquivalentTracelsTerTrace : {lu : LUniv}{c : Choice}{x : Maybe (ChoiceSet c)}{a : ChoiceSet c}
  (P : Process ∞ {lu} c)(l : List (Label lu))
  (terequivP : TerminateEquivalent a P)(tr : Tr l x P)
  → Tr l x (terminate a)
termEquivalentTracelsTerTrace {lu}{c} {x} {a} .(terminate a) l (termeqterm tr) = tr
termEquivalentTracelsTerTrace {lu}{c} {x} {a} .(node P) l (termeqnode {P} terequivQ) (tnode tr) =
  termEquivalentTracelsTerTrace+ P l (termeqnode terequivQ)

```

mutual

```

traceAppendTrw∞ : {lu : LUniv}(c : Choice)(P : Process ∞ {lu} c)(Q : Process ∞ {lu} c)
  (l1 l2 : List (Label lu))(m : Maybe (ChoiceSet c))
  (tr1 : P →∞*[ l1 ] Q)(tr2 : Tr l2 m Q)
  → Tr∞ (l1 ++ l2) m P
traceAppendTrw∞ c P Q l1 l2 m tr1 tr2 = traceAppendTrw c (forcep P) Q l1 l2 m tr1 tr2

traceAppendTrw : {lu : LUniv}(c : Choice)(P : Process ∞ {lu} c)
  (Q : Process ∞ {lu} c)(l1 l2 : List (Label lu))(m : Maybe (ChoiceSet c))
  (tr1 : P →*[ l1 ] Q)(tr2 : Tr l2 m Q)
  → Tr (l1 ++ l2) m P
traceAppendTrw c .(terminate x) .(terminate x) .[] l2 m (empty x) tr2 = tr2
traceAppendTrw c .(node P) .(node P) .[] l2 m (tnode {[]} {.(inj1 (node P))}) {P} (empty) tr2 = tr2
traceAppendTrw c .(node P) Q .(Lab P x :: l) l2 m (tnode {.(Lab P x :: l)} {.(inj1 Q)}) {P} (extc l .(i
  = tnode (extc (l ++ l2) m x (traceAppendTrw∞ c (PE P x) Q l l2 m tr tr2))
traceAppendTrw c .(node P) Q l1 l2 m (tnode {l1} {.(inj1 Q)}) {P} (intc .l1 .(inj1 Q) x tr)) tr2
  = tnode (intc (l1 ++ l2) m x (traceAppendTrw∞ c (PI P x) Q l1 l2 m tr tr2))

traceAppendTrw+ : {lu : LUniv}(c : Choice)(P : Process+ ∞ {lu} c)(Q : Process ∞ {lu} c)
  (l1 l2 : List (Label lu))(m : Maybe (ChoiceSet c))
  (tr1 : P →+*[ l1 ] Q)(tr2 : Tr l2 m Q)
  → Tr+ (l1 ++ l2) m P
traceAppendTrw+ c P .(node P) .[] l2 m empty (tnode tr) = tr
traceAppendTrw+ c P Q .(Lab P x :: l) l2 m (extc l .(inj1 Q) x x1) tr2
  = extc (l ++ l2) m x (traceAppendTrw∞ c (PE P x) Q l l2 m x1 tr2)
traceAppendTrw+ c P Q l1 l2 m (intc .l1 .(inj1 Q) x x1) tr2
  = intc (l1 ++ l2) m x (traceAppendTrw∞ c (PI P x) Q l1 l2 m x1 tr2)

```

$$\begin{aligned} \text{trPAppendTrw}_{\infty} : & \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process}_{\infty} \infty \{lu\} \ c)(Q : \text{Process} \infty \{lu\} \ c) \\ & (l_1 \ l_2 : \text{List}(\text{Label } lu))(m : \text{Process} \infty \{lu\} \ c \uplus \text{ChoiceSet } c) \\ & (tr_1 : P \rightarrow_{\infty}^{*} [l_1] \ Q)(tr_2 : \text{TrP } l_2 \ m \ Q) \\ & \rightarrow \text{TrP}_{\infty} (l_1 ++ l_2) \ m \ P \end{aligned}$$

$$\begin{aligned} \text{trAppendTrw} : & \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process} \infty \{lu\} \ c)(Q : \text{Process} \infty \{lu\} \ c) \\ & (l_1 \ l_2 : \text{List} \ (\text{Label} \ lu))(m : \text{Process} \infty \{lu\} \ c \uplus \text{ChoiceSet} \ c) \\ & (tr_1 : P \rightarrow^* [l_1] \ Q)(tr_2 : \text{TrP} \ l_2 \ m \ Q) \\ & \rightarrow \text{TrP} \ (l_1 ++ l_2) \ m \ P \end{aligned}$$

$$\begin{aligned} \text{trPAppendTrw+} : & \{lu : \text{LUniv}\}(c : \text{Choice})(P : \text{Process+} \infty \{lu\} \ c)(Q : \text{Process} \infty \{lu\} \ c) \\ & (l_1 \ l_2 : \text{List} \ (\text{Label} \ lu))(m : \text{Process} \infty \{lu\} \ c \uplus \text{ChoiceSet} \ c) \\ & (tr_1 : P \rightarrow +^* [l_1] \ Q)(tr_2 : \text{TrP} \ l_2 \ m \ Q) \\ & \rightarrow \text{TrP+} \ (l_1 \ ++ \ l_2) \ m \ P \end{aligned}$$

$$\begin{aligned} \text{trPResultToTrResult} : & \{lu : \text{LUniv}\} \{c : \text{Choice}\} (l : \text{List} (\text{Label } lu)) \\ & (mc : \text{Process } \infty \{lu\} \text{ } c \uplus \text{ChoiceSet } c) \\ & \rightarrow \text{Maybe} (\text{ChoiceSet } c) \end{aligned}$$

$$\begin{aligned} \text{trPResultToTrResult } \{lu\}\{c\} \ l \ (\text{inj}_1 \ -) \\ &= \text{nothing} \\ \text{trPResultToTrResult } \{lu\}\{c\} \ l \ (\text{inj}_2 \ x) &= \text{just } x \end{aligned}$$

```

trPtoTr $\infty$  : {lu : LUniv}{c : Choice } (l : List (Label lu)) (mc : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (P : Process $\infty$   $\infty$  {lu} c) (tr : TrP $\infty$  {lu}{c} l mc P)
   $\rightarrow$  Tr $\infty$  {lu}{c} l (trPResultToTrResult {lu}{c} l mc) P
trPtoTr $\infty$  l mc P tr
  = trPtoTr l mc (forcep P) tr

trPtoTr : {lu : LUniv}{c : Choice } (l : List (Label lu)) (mc : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (P : Process  $\infty$  {lu} c) (tr : TrP {lu}{c} l mc P)
   $\rightarrow$  Tr {lu}{c} l (trPResultToTrResult {lu}{c} l mc) P

trPtoTr .[] .(inj2 x) .(terminate x) (ter x) = ter x
trPtoTr .[] .(inj1 (terminate x)) .(terminate x) (empty x) = empty x
trPtoTr .[] .(inj1 (node P)) .(node P) (tnode {[]} {(inj1 (node P))}) {P} empty
  = tnode empty
trPtoTr .(Lab P x :: l) (inj1 (terminate x1)) .(node P) (tnode {(Lab P x :: l)}
  {(inj1 (terminate x1))} {P} (extc l .(inj1 (terminate x1)) x x2))
  = tnode (extc l (trPResultToTrResult (Lab P x :: l) (inj1 (terminate x1))) x (trPtoTr $\infty$  l (inj1 (-)) (PE P x) x2))
trPtoTr .(Lab P x :: l) (inj1 (node x1)) .(node P) (tnode {(Lab P x :: l)} {(inj1 (node x1))} {P} (extc l .(inj1 (node x1)) x x2))
  = tnode (extc l (trPResultToTrResult (Lab P x :: l) (inj1 (node x1))) x (trPtoTr $\infty$  l (inj1 (-)) (PE P x) x2))
trPtoTr .(Lab P x :: l) (inj2 y) .(node P) (tnode {(Lab P x :: l)} {(inj2 y)} {P} (extc l .(inj2 y) x x1))
  = tnode (extc l (trPResultToTrResult (Lab P x :: l) (inj2 y)) x (trPtoTr $\infty$  l (inj2 y) (PE P x) x1))
trPtoTr l mc .(node P) (tnode {l} {mc} {P} (intc l .mc x x1))
  = tnode (intc l (trPResultToTrResult l mc) x (trPtoTr $\infty$  l mc (PI P x) x1))
trPtoTr .[] .(inj2 (PT P x)) .(node P) (tnode {[]} {(inj2 (PT P x))} {P} (terc x))
  = tnode (terc x)

trPtoTr+ : {lu : LUniv}{c : Choice } (l : List (Label lu)) (mc : Process  $\infty$  {lu} c  $\uplus$  ChoiceSet c)
  (P : Process+  $\infty$  {lu} c) (tr : TrP+ {lu}{c} l mc P)
   $\rightarrow$  Tr+ {lu}{c} l (trPResultToTrResult {lu}{c} l mc) P
trPtoTr+ .[] .(inj1 (node P)) P empty
  = empty
trPtoTr+ .(Lab P x :: l) (inj1 (terminate x1)) P (extc l .(inj1 (terminate x1)) x x2)
  = extc l (trPResultToTrResult (Lab P x :: l) (inj1 (terminate x1))) x (trPtoTr $\infty$  l (inj1 (-)) (PE P x) x2)
trPtoTr+ .(Lab P x :: l) (inj1 (node x1)) P (extc l .(inj1 (node x1)) x x2)
  = extc l (trPResultToTrResult (Lab P x :: l) (inj1 (node x1))) x (trPtoTr $\infty$  l (inj1 (-)) (PE P x) x2)
trPtoTr+ .(Lab P x :: l) (inj2 y) P (extc l .(inj2 y) x x1)
  = extc l (trPResultToTrResult (Lab P x :: l) (inj2 y)) x (trPtoTr $\infty$  l (inj2 y) (PE P x) x1)
trPtoTr+ l mc P (intc l .mc x x1)
  = intc l (trPResultToTrResult l mc) x (trPtoTr $\infty$  l mc (PI P x) x1)
trPtoTr+ .[] .(inj2 (PT P x)) P (terc x)
  = terc x

```

A.114 TraceWithNextProcess.agda

```
--@PREFIX@mainTraceWithNextProcess
```

```
module TraceWithNextProcess where
```

```
open import Size
open import Data.List
open import Data.Product
open import Data.Maybe
open import labelUniv
open import process
open import choiceSetU
open import Data.Sum
```

```
mutual
```

```
--@BEGIN@TrPPlusDef
```

```
data TrP+ {lu : LUniv}{c : Choice} : (l : List (Label lu))
  → Process ∞ {lu} c ⊔ ChoiceSet c
  → (P : Process+ ∞ {lu} c) → Set where
empty : {P : Process+ ∞ {lu} c} → TrP+ {lu} [] (inj1 (node P)) P
extc   : {P : Process+ ∞ {lu} c}
  → (l : List (Label lu))
  → (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (x : ChoiceSet (E P))
  → TrP∞ {lu} l tick (PE P x)
  → TrP+ {lu} (Lab P x :: l) tick P
intc   : {P : Process+ ∞ {lu} c}
  → (l : List (Label lu))
  → (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (x : ChoiceSet (I P))
  → TrP∞ {lu} l tick (PI P x)
  → TrP+ {lu} l tick P
terc   : {P : Process+ ∞ {lu} c}
  → (x : ChoiceSet (T P))
  → TrP+ {lu} [] (inj2 (PT P x)) P
```

```

--@END

--@BEGIN@TrPDef

data TrP {lu : LUniv}{c : Choice} (l : List (Label lu))
  → Process ∞ {lu} c ⊔ ChoiceSet c
  → (P : Process ∞ {lu} c) → Set where
  ter      : (x : ChoiceSet c) → TrP {lu} [] (inj2 x) (terminate x)
  empty    : (x : ChoiceSet c) → TrP {lu} [] (inj1 (terminate x)) (terminate x)
  tnode    : {l : List (Label lu)}
    → {x : Process ∞ {lu} c ⊔ ChoiceSet c}
    → {P : Process+ ∞ {lu} c}
    → TrP+ {lu} {c} l x P
    → TrP {lu} l x (node P)

--@END

--@BEGIN@TrPInfDef

TrP∞ : {lu : LUniv}{c : Choice} (l : List (Label lu))
  (tick : Process ∞ {lu} c ⊔ ChoiceSet c)
  (P : Process∞ ∞ {lu} c) → Set
TrP∞ {lu} {c} l tick P = TrP {lu} l tick (forcep P)

--@END

_→∞*[_]_ : {lu : LUniv}{c : Choice}(P : Process∞ ∞ {lu} c)
  (l : List (Label lu))
  (Q : Process ∞ {lu} c) → Set
_→∞*[_]_ {lu} {c} P l Q = TrP∞ {lu} {c} l (inj1 Q) P
-- P →∞*[ 1 ] Q = TrP∞ {lu} 1 (inj1 Q) P

_→*[_]_ : {lu : LUniv}{c : Choice}(P : Process ∞ {lu} c)
  (l : List (Label lu))
  (Q : Process ∞ {lu} c) → Set
_→*[_]_ {lu} {c} P l Q = TrP {lu} {c} l (inj1 Q) P
-- P →*[ 1 ] Q = TrP {lu} 1 (inj1 Q) P

--@BEGIN@Trsyntacs

_→+*[_]_ : {lu : LUniv}{c : Choice}(P : Process+ ∞ {lu} c)

```

```

      (l : List (Label lu))
      (Q : Process ∞ {lu} c) → Set
    _→+*[_]_ {lu} {c} P l Q = TrP+ {lu} {c} l (inj1 Q) P

--@END

-- P →+*[ 1 ] Q = TrP+ {lu} 1 (inj1 Q) P

TrP+2c : {lu : LUniv} {c : Choice} (l : List (Label lu))
  → (m : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (P : Process+ ∞ {lu} c)
  → TrP+ {lu} l m P
  → Choice
TrP+2c {lu} {c} l m P x = E P

TrP+2cs : {lu : LUniv} {c : Choice} (l : List (Label lu))
  → (m : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (P : Process+ ∞ {lu} c)
  → TrP+ {lu} l m P
  → Set
TrP+2cs {lu} {c} l m P x = ChoiceSet (TrP+2c {lu} l m P x)

TrP+2P : {lu : LUniv} {c : Choice} (l : List (Label lu))
  → (m : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (P : Process+ ∞ {lu} c)
  → TrP+ {lu} l m P
  → Process+ ∞ {lu} c
TrP+2P {lu} {c} l m P x = P

TrP+2Q : {lu : LUniv} {c : Choice} (l : List (Label lu))
  → (m : Process ∞ {lu} c ⊔ ChoiceSet c)
  → (P : Process+ ∞ {lu} c)
  → TrP+ {lu} l m P
  → Process ∞ {lu} c

```

$\text{TrP}+2\text{Q} \{lu\} \{c\} l (\text{inj}_1 x) P \ x_1 = x$
 $\text{TrP}+2\text{Q} \{lu\} \{c\} l (\text{inj}_2 y) P \ x = \text{terminate } y$

$\text{forcetP}' : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (l : \text{List } (\text{Label } lu))$
 $\quad (tick : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\quad \rightarrow (P : \text{Process}+ \infty \{lu\} c)$
 $\quad \rightarrow \text{TrP} \{lu\} \{c\} l \text{ tick } (\text{node } P)$
 $\quad \rightarrow \text{TrP}+ \{lu\} \{c\} l \text{ tick } P$
 $\text{forcetP}' l \text{ tick } P (\text{tnode } q) = q$

$\text{delayP} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (l : \text{List } (\text{Label } lu))$
 $\quad (tick : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\quad \rightarrow \{P : \text{Process}+ \infty \{lu\} c\}$
 $\quad \rightarrow \text{TrP}+ \{lu\} \{c\} l \text{ tick } P$
 $\quad \rightarrow \text{TrP} \infty \{lu\} \{c\} l \text{ tick } (\text{delay } (\text{node } P))$
 $\text{delayP} \{c\} l \text{ tick } \{P\} p = \text{tnode } p$

$\text{forcetP} \infty p : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (l : \text{List } (\text{Label } lu))$
 $\quad (tick : \text{Process} \infty \{lu\} c \uplus \text{ChoiceSet } c)$
 $\quad \rightarrow \{P : \text{Process} \infty \infty \{lu\} c\}$
 $\quad \rightarrow \text{TrP} \infty \{lu\} \{c\} l \text{ tick } P$
 $\quad \rightarrow \text{TrP} \{lu\} \{c\} l \text{ tick } (\text{forcep } P)$
 $\text{forcetP} \infty p \{c\} l \text{ tick } \{P\} x = x$

mutual

$\text{reflTrP} : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \{lu\} c)$
 $\quad \rightarrow \text{TrP} \{lu\} [] (\text{inj}_1 P) P$
 $\text{reflTrP} \{c\} (\text{terminate } x) = \text{empty } x$
 $\text{reflTrP} \{c\} (\text{node } Q) = \text{tnode empty}$

$\text{reflTrP} \infty : \{lu : \text{LUniv}\} \{c : \text{Choice}\} (P : \text{Process} \infty \infty \{lu\} c)$
 $\quad \rightarrow \text{TrP} \infty \{lu\} [] (\text{inj}_1 (\text{forcep } P)) P$

```

    reflTrP $\infty$  P = reflTrP (forcep P)

--@BEGIN@refTrPDef

 $\_ \infty \sqsubseteq p \_$  : {lu : LUniv}{c : Choice} (P : Process $\infty$  {lu} c)
              (Q : Process $\infty$  {lu} c)  $\rightarrow$  Set
 $\_ \infty \sqsubseteq p \_$  {lu}{c} P Q = (l : List (Label lu))
                         $\rightarrow$  (m : Process $\infty$  {lu} c  $\uplus$  ChoiceSet c)
                         $\rightarrow$  TrP {lu} l m Q  $\rightarrow$  TrP {lu} l m P

--@END

 $\infty$ Refp =  $\_ \infty \sqsubseteq p \_$ 

--@BEGIN@refTrPInfDef

 $\_ \infty \sqsubseteq \infty \_$  : {lu : LUniv}{c : Choice}  $\rightarrow$  (P : Process $\infty$  {lu} c)
              (Q : Process $\infty$  {lu} c)  $\rightarrow$  Set
 $\_ \infty \sqsubseteq \infty \_$  {lu}{c} P Q = (l : List (Label lu))
                         $\rightarrow$  (m : Process $\infty$  {lu} c  $\uplus$  ChoiceSet c)
                         $\rightarrow$  TrP $\infty$  {lu} l m Q  $\rightarrow$  TrP $\infty$  {lu} l m P

--@END

 $\infty$ Ref $\infty$  =  $\_ \infty \sqsubseteq \infty \_$ 

--@BEGIN@refTrPPlusDef

 $\_ \infty \sqsubseteq + \_$  : {lu : LUniv}{c : Choice} (P : Process+ {lu} c)
              (Q : Process+ {lu} c)  $\rightarrow$  Set
 $\_ \infty \sqsubseteq + \_$  {lu}{c} P Q = (l : List (Label lu))
                         $\rightarrow$  (m : Process $\infty$  {lu} c  $\uplus$  ChoiceSet c)
                         $\rightarrow$  TrP+ {lu} l m Q  $\rightarrow$  TrP+ {lu} l m P

 $\infty$ Ref+ =  $\_ \infty \sqsubseteq + \_$ 

--@END

```

```

--@BEGIN@EqTrPPPlusDef

_∞≡p_ : {lu : LUniv}{c : Choice} → (P Q : Process ∞ {lu} c) → Set
P ∞≡p Q = (P ∞⊑p Q) × ( Q ∞⊑p P)

--@END

--@BEGIN@ProofLawTrPDef

refl∞⊑p : {lu : LUniv}{c : Choice} → (P : Process ∞ {lu} c) → P ∞⊑p P
refl∞⊑p P l m x = x

antiSym∞⊑p : {lu : LUniv}{c : Choice} → (P Q : Process ∞ {lu} c) → P ∞⊑p Q
→ Q ∞⊑p P → P ∞≡p Q
antiSym∞⊑p P Q PQ QP = PQ , QP

trans∞⊑p : {lu : LUniv}{c : Choice} → (P Q R : Process ∞ {lu} c) → P ∞⊑p Q
→ Q ∞⊑p R → P ∞⊑p R
trans∞⊑p P Q R PQ QR l m tr = PQ l m (QR l m tr)

--@END

```

A.115 TraceWithoutSize.agda

```

--@PREFIX@traceWithoutSize

module TraceWithoutSize where

open import Size
open import Data.List
open import Data.Product
open import Data.Maybe
open import process
open import choiceSetU
open import labelUniv

```

```
--@BEGIN@TrDefplus
```

```
mutual
```

```
data Tr+ {lu : LUniv}{c : Choice } : (l : List (Label lu))
    → (m : Maybe (ChoiceSet c))
    → (P : Process+ ∞ {lu} c) → Set where
empty : {P : Process+ ∞ {lu} c} → Tr+ [] nothing P
extc   : {P : Process+ ∞ {lu} c}
    → (l : List (Label lu))
    → (mc : Maybe (ChoiceSet c))
    → (x : ChoiceSet (E P))
    → (tr : Tr∞ {lu} l mc (PE P x))
    → Tr+ {lu} (Lab P x :: l) mc P
intc   : {P : Process+ ∞ {lu} c}
    → (l : List (Label lu))
    → (mc : Maybe (ChoiceSet c))
    → (x : ChoiceSet (I P))
    → (tr : Tr∞ {lu} l mc (PI P x))
    → Tr+ {lu} l mc P
terc   : {P : Process+ ∞ {lu} c}
    → (t : ChoiceSet (T P))
    → Tr+ {lu} [] (just (PT P t)) P
```

```
--@END
```

```
--@BEGIN@TrDef
```

```
data Tr {lu : LUniv}{c : Choice } : (l : List (Label lu))
    (m : Maybe (ChoiceSet c))
    (P : Process ∞ {lu} c) → Set where
ter    : (x : ChoiceSet c) → Tr {lu} [] (just x) (terminate x)
empty  : (x : ChoiceSet c) → Tr {lu} [] nothing (terminate x)
tnode  : {l : List (Label lu)}
    → {x : Maybe (ChoiceSet c)}
    → {P : Process+ ∞ {lu} c}
    → (tr : Tr+ {lu} {c} l x P)
    → Tr {lu} l x (node P)
```

```
--@END
```

```
--@BEGIN@TrInfDef
```

```

Tr $\infty$  : {lu : LUniv}{c : Choice} (l : List (Label lu))
      (tick : Maybe (ChoiceSet c))
      (P : Process $\infty$   $\infty$  {lu} c)  $\rightarrow$  Set
Tr $\infty$  {c} l tick P = Tr l tick (forcep P)

--@END

```

A.116 trainExample.agda

```

--@PREFIX@trainExample

module trainExample where

open import Data.Bool hiding (_==_)
open import Data.Sum
open import Data.Maybe
open import Data.Bool.Base renaming (T to T') hiding (_==_)
open import libBool
open import auxData
open import libList
open import libEq
open import Data.String
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List      renaming (_++_ to _++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin
open import Data.Nat hiding (_==_)
open import parallelSimple

```

```

open import choiceSetU
open import interleave
open import hidingOperator
open import renamingOperator
open import primitiveProcess
open import TraceWithoutSize
open import RefWithoutSize
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule

```

```

-----
-----
-----SEGMENT-----
-----
-----

```

```

--@BEGIN@SEGMENT
data SEGMENT : Set where   seg1 : SEGMENT
--@END

```

```

_==segm_ : SEGMENT → SEGMENT → Bool
seg1 ==segm seg1 = true

```

```

refl==segm : (s : SEGMENT) → T' (s ==segm s)
refl==segm seg1 = _

```

```

sym==segm : (s s' : SEGMENT) → T' (s ==segm s') → T' (s' ==segm s)
sym==segm seg1 seg1 _ = _

```

```

transfsegm : (s s' : SEGMENT) (Q : SEGMENT → Set)
  → T' (s ==segm s')
  → Q s → Q s'
transfsegm seg1 seg1 Q _      q = q

```

```
showSEGMENT : (s : SEGMENT) → String
```

```
showSEGMENT sig1 = "sig1"
```

```
LabelListSEGMENT : List SEGMENT
```

```
LabelListSEGMENT = sig1 :: []
```

```
labelSEGMENT : LUniv
```

```
LUniv.Labelf labelSEGMENT = SEGMENT
```

```
LUniv._==If_ labelSEGMENT = _==segm_
```

```
LUniv.refl==If labelSEGMENT {s} = refl==segm s
```

```
LUniv.showLabelf labelSEGMENT = showSEGMENT
```

```
LUniv.LabelListf labelSEGMENT = LabelListSEGMENT
```

```
LUniv.sym==If labelSEGMENT {s} {s'} = sym==segm s s'
```

```
LUniv.transf labelSEGMENT {s} {s'} = transfsegm s s'
```

```
-----
-----
-----SIGNAL-----
-----
-----
```

```
--@BEGIN@SIGNAL
```

```
data SIGNAL : Set where sig1      sig2 : SIGNAL
```

```
--@END
```

```
_==sig_ : SIGNAL → SIGNAL → Bool
```

```
sig1 ==sig sig1 =      true
```

```
sig2 ==sig sig2 =      true
```

```
_ ==sig _ =      false
```

```
refl==sig : (s : SIGNAL) → T' (s ==sig s)
```

```
refl==sig sig1 = _
```

```
refl==sig sig2 = _
```

```
showSIGNAL : (s : SIGNAL) → String
```

```
showSIGNAL sig1 = "sig1"
```

```
showSIGNAL sig2 = "sig2"
```

```

sym==sig : (s s' : SIGNAL) → T' (s ==sig s') → T' (s' ==sig s)
sym==sig sig1 sig1 _ = _
sym==sig sig1 sig2 ()
sym==sig sig2 sig1 ()
sym==sig sig2 sig2 _ = _

```

```

transfsig : (s s' : SIGNAL) (Q : SIGNAL → Set)
           → T' (s ==sig s')
           → Q s → Q s'
transfsig sig1 sig1 Q _      q = q
transfsig sig1 sig2 Q ()
transfsig sig2 sig1 Q ()
transfsig sig2 sig2 Q _      q = q

```

```

LabelListSIGNAL : List SIGNAL
LabelListSIGNAL = sig1 :: sig2 :: []

```

```

labelSIGNAL : LUniv
LUniv.Labelf labelSIGNAL = SIGNAL
LUniv._==If_ labelSIGNAL = _==sig_
LUniv.refl==If labelSIGNAL {l} = refl==sig l
LUniv.showLabelf labelSIGNAL = showSIGNAL
LUniv.LabelListf labelSIGNAL = LabelListSIGNAL
LUniv.sym==If labelSIGNAL {s} {s'} = sym==sig s s'
LUniv.transf labelSIGNAL {s} {s'} = transfsig s s'

```

```

-----
-----
-----TRAIN-----
-----
-----

```

```

--@BEGIN@TRAIN
data TRAIN : Set where      ta tb : TRAIN
--@END
_==train_ : TRAIN → TRAIN → Bool

```

```

ta ==train ta =      true
tb ==train tb =      true
_  ==train _  =      false

```

```

refl==train : (s : TRAIN) → T' (s ==train s)
refl==train ta = _
refl==train tb = _

```

```

sym==train : (t t' : TRAIN) → T' (t ==train t') → T' (t' ==train t)
sym==train ta ta _ = _
sym==train ta tb ()
sym==train tb ta ()
sym==train tb tb _ = _

```

```

transftrain : (t t' : TRAIN) (Q : TRAIN → Set)
              → T' (t ==train t')
              → Q t → Q t'
transftrain ta ta Q _      q = q
transftrain ta tb Q ()
transftrain tb ta Q ()
transftrain tb tb Q _      q = q

```

```

showTRAIN : (s : TRAIN) → String
showTRAIN ta = "ta"
showTRAIN tb = "tb"

```

```

LabelListTRAIN : List TRAIN
LabelListTRAIN = ta :: tb :: []

```

```

labelTRAIN : LUniv
LUniv.Labelf labelTRAIN = TRAIN
LUniv._==If_ labelTRAIN = _==train_
LUniv.refl==If labelTRAIN {l} = refl==train l
LUniv.showLabelf labelTRAIN = showTRAIN
LUniv.LabelListf labelTRAIN = LabelListTRAIN
LUniv.sym==If labelTRAIN {t} {t'} = sym==train t t'
LUniv.transf labelTRAIN {t} {t'} = transftrain t t'

```

-----ASPECT-----

--@BEGIN@ASPECT

data ASPECT : Set where red green : ASPECT

--@END

==aspect : ASPECT → ASPECT → Bool

red ==aspect red = true

green ==aspect green = true

_ ==aspect _ = false

refl==aspect : (s : ASPECT) → T' (s ==aspect s)

refl==aspect red = _

refl==aspect green = _

sym==aspect : (a a' : ASPECT) → T' (a ==aspect a') → T' (a' ==aspect a)

sym==aspect green green _ = _

sym==aspect green red ()

sym==aspect red green ()

sym==aspect red red _ = _

transaspect : (a a' : ASPECT) (Q : ASPECT → Set)

→ T' (a ==aspect a')

→ Q a → Q a'

transaspect green green Q _ q = q

transaspect green red Q ()

transaspect red green Q ()

transaspect red red Q _ q = q

showASPECT : (s : ASPECT) → String

showASPECT red = "red"

showASPECT green = "green"

LabelListASPECT : List ASPECT

LabelListASPECT = red :: green :: []

```

labelASPECT : LUniv
LUniv.Labelf labelASPECT = ASPECT
LUniv._==If_ labelASPECT = _==aspect_
LUniv.refl==If labelASPECT {l} = refl==aspect l
LUniv.showLabelf labelASPECT = showASPECT
LUniv.LabelListf labelASPECT = LabelListASPECT
LUniv.sym==If labelASPECT {a} {a'} = sym==aspect a a'
LUniv.transf labelASPECT {a} {a'} = transfaspect a a'

```

```

-----
-----
-----SEGSTATE-----
-----
-----

```

```

--@BEGIN@SEGSTATE
data SEGSTATE : Set where free   blocked : SEGSTATE
--@END

```

```

_==segstate_ : SEGSTATE → SEGSTATE → Bool
free ==segstate free = true
blocked ==segstate blocked = true
_ ==segstate _ = false

```

```

refl==segstate : (s : SEGSTATE) → T' (s ==segstate s)
refl==segstate free = _
refl==segstate blocked = _

```

```

sym==segstate : (s s' : SEGSTATE) → T' (s ==segstate s') → T' (s' ==segstate s)
sym==segstate free free _ = _
sym==segstate free blocked ()
sym==segstate blocked free ()
sym==segstate blocked blocked _ = _

```

```

transfsegstate : (s s' : SEGSTATE) (Q : SEGSTATE → Set)
  → T' (s ==segstate s')
  → Q s → Q s'
transfsegstate free free Q _      q = q

```

```

transfsegstate free blocked Q ()
transfsegstate blocked free Q ()
transfsegstate blocked blocked Q _      q = q

showSEGSTATE : (s : SEGSTATE) → String
showSEGSTATE free = "free"
showSEGSTATE blocked = "blocked"

LabelListSEGSTATE : List SEGSTATE
LabelListSEGSTATE = free :: blocked :: []

labelSEGSTATE : LUniv
LUniv.Labelf labelSEGSTATE = SEGSTATE
LUniv._==If_ labelSEGSTATE = _==segstate_
LUniv.refl==If labelSEGSTATE {l} = refl==segstate l
LUniv.showLabelf labelSEGSTATE = showSEGSTATE
LUniv.LabelListf labelSEGSTATE = LabelListSEGSTATE
LUniv.sym==If labelSEGSTATE {s} {s'} = sym==segstate s s'
LUniv.transf labelSEGSTATE {s} {s'} = transfsegstate s s'

-----LabelTrains-----

--@BEGIN@LabelTrains
data LabelTrains : Set where
  getSegm : TRAIN → SEGMENT → SEGSTATE → LabelTrains
--@END

--@BEGIN@LabelsetSegm
  setSegm : TRAIN → SEGMENT → SEGSTATE → LabelTrains
--@END

--@BEGIN@LabelsetSig
  setSig : TRAIN → SIGNAL → ASPECT → LabelTrains
--@END

--@BEGIN@LabelsetSigs
  setSigs : SIGNAL → ASPECT → LabelTrains

```

--@END

```

_==LabelTrains_ : LabelTrains → LabelTrains → Bool
getSegm t seg segs      ==LabelTrains getSegm t' seg' segs'  =
  ==Triple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    (t ,, (seg ,, segs)) (t' ,, (seg' ,, segs'))
setSegm t seg segs      ==LabelTrains setSegm t' seg' segs'  =
  ==Triple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    (t ,, (seg ,, segs)) (t' ,, (seg' ,, segs'))
setSig t sig asp        ==LabelTrains setSig t' sig' asp'      =
  ==Triple {TRAIN} {SIGNAL} {ASPECT} _==train_ _==sig_ _==aspect_
    (t ,, (sig ,, asp)) (t' ,, (sig' ,, asp'))
setSigs sig asp         ==LabelTrains setSigs sig' asp'       =
  ==Pair {SIGNAL} {ASPECT} _==sig_ _==aspect_
    (sig ,, asp) (sig' ,, asp')
_ ==LabelTrains _ = false

```

```

refl==labelTrains : (l : LabelTrains) → T' (l ==LabelTrains l)
refl==labelTrains (getSegm t seg segst) =
  reflTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    refl==train refl==segm refl==segstate (t ,, (seg ,, segst))
refl==labelTrains (setSegm t seg segst) =
  reflTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    refl==train refl==segm refl==segstate (t ,, (seg ,, segst))
refl==labelTrains (setSig t sig asp) =
  reflTriple {TRAIN} {SIGNAL} {ASPECT} _==train_ _==sig_ _==aspect_
    refl==train refl==sig refl==aspect (t ,, (sig ,, asp))
refl==labelTrains (setSigs sig asp) =
  reflPair {SIGNAL} {ASPECT} _==sig_ _==aspect_
    refl==sig refl==aspect (sig ,, asp)

```

```

sym==labelTrains : (l l' : LabelTrains) → T' (l ==LabelTrains l')
  → T' (l' ==LabelTrains l)
sym==labelTrains (getSegm t seg segst) (getSegm t' seg' segst') =
  symTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    sym==train sym==segm sym==segstate
      (t ,, (seg ,, segst)) (t' ,, (seg' ,, segst'))
sym==labelTrains (setSegm t seg segst) (setSegm t' seg' segst') =
  symTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_ _==segstate_
    sym==train sym==segm sym==segstate

```

```

      (t „ (seg „ segst)) (t' „ (seg' „ segst'))
sym==labelTrains (setSig t sig asp) (setSig t' sig' asp')      =
  symTriple {TRAIN} {SIGNAL} {ASPECT} _==train_ _==sig_ _==aspect_
  sym==train sym==sig sym==aspect
  (t „ (sig „ asp)) (t' „ (sig' „ asp'))
sym==labelTrains (setSigs sig asp) (setSigs sig' asp')      =
  symPair {SIGNAL} {ASPECT} _==sig_ _==aspect_
  sym==sig sym==aspect
  (sig „ asp) (sig' „ asp')
sym==labelTrains (getSegm x x1 x2) (setSegm x3 x4 x5) ()
sym==labelTrains (getSegm x x1 x2) (setSig x3 x4 x5) ()
sym==labelTrains (getSegm x x1 x2) (setSigs x3 x4) ()
sym==labelTrains (setSegm x x1 x2) (getSegm x3 x4 x5) ()
sym==labelTrains (setSegm x x1 x2) (setSig x3 x4 x5) ()
sym==labelTrains (setSegm x x1 x2) (setSigs x3 x4) ()
sym==labelTrains (setSig x x1 x2) (getSegm x3 x4 x5) ()
sym==labelTrains (setSig x x1 x2) (setSegm x3 x4 x5) ()
sym==labelTrains (setSig x x1 x2) (setSigs x3 x4) ()
sym==labelTrains (setSigs x x1) (getSegm x2 x3 x4) ()
sym==labelTrains (setSigs x x1) (setSegm x2 x3 x4) ()
sym==labelTrains (setSigs x x1) (setSig x2 x3 x4) ()

translabelTrains : (l l' : LabelTrains) (Q : LabelTrains → Set)
  → T' (l ==LabelTrains l')
  → Q l → Q l'
translabelTrains (getSegm t seg segst) (getSegm t' seg' segst') Q =
  transTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_
  _==segstate_ transftrain transfsegm transfsegstate
  (t „ (seg „ segst)) (t' „ (seg' „ segst'))
  (λ {(t „ (seg „ segst))} → Q (getSegm t seg segst)})
translabelTrains (setSegm t seg segst) (setSegm t' seg' segst') Q =
  transTriple {TRAIN} {SEGMENT} {SEGSTATE} _==train_ _==segm_
  _==segstate_ transftrain transfsegm transfsegstate
  (t „ (seg „ segst)) (t' „ (seg' „ segst'))
  (λ {(t „ (seg „ segst))} → Q (setSegm t seg segst)})
translabelTrains (setSig t sig asp) (setSig t' sig' asp') Q =
  transTriple {TRAIN} {SIGNAL} {ASPECT} _==train_ _==sig_
  _==aspect_ transftrain transfsig transfaspect
  (t „ (sig „ asp)) (t' „ (sig' „ asp'))
  (λ {(t „ (sig „ asp))} → Q (setSig t sig asp)})
translabelTrains (setSigs sig asp) (setSigs sig' asp') Q =

```

```

transfPair {SIGNAL} {ASPECT} _==sig_ _==aspect_
  transfsig transfaspect
  (sig „ asp) (sig' „ asp')
  (λ {(sig „ asp)      → Q (setSigs sig asp)})
translabelTrains (getSegm x x1 x2) (setSegm x3 x4 x5) Q ()
translabelTrains (getSegm x x1 x2) (setSig x3 x4 x5) Q ()
translabelTrains (getSegm x x1 x2) (setSigs x3 x4) Q ()
translabelTrains (setSegm x x1 x2) (getSegm x3 x4 x5) Q ()
translabelTrains (setSegm x x1 x2) (setSig x3 x4 x5) Q ()
translabelTrains (setSegm x x1 x2) (setSigs x3 x4) Q ()
translabelTrains (setSig x x1 x2) (getSegm x3 x4 x5) Q ()
translabelTrains (setSig x x1 x2) (setSegm x3 x4 x5) Q ()
translabelTrains (setSig x x1 x2) (setSigs x3 x4) Q ()
translabelTrains (setSigs x x1) (getSegm x2 x3 x4) Q ()
translabelTrains (setSigs x x1) (setSegm x2 x3 x4) Q ()
translabelTrains (setSigs x x1) (setSig x2 x3 x4) Q ()

```

```

showLabelTrains : (s : LabelTrains) → String
showLabelTrains (getSegm x x1 x2) = showTRAIN    x ++s showSEGMENT x1 ++s showSEGSTATE x2
showLabelTrains (setSegm x x1 x2) = showTRAIN    x ++s showSEGMENT x1 ++s showSEGSTATE x2
showLabelTrains (setSig x x1 x2)      = showTRAIN x ++s showSIGNAL   x1 ++s showASPECT  x2
showLabelTrains (setSigs x x1)         = showSIGNAL x ++s showASPECT x1

```

```

LabelListLabelTrainsGetSegm : List LabelTrains
LabelListLabelTrainsGetSegm = LUnion LabelListTRAIN      λ tr →
  LUnion LabelListSEGMENT λ segm →
  LUnion LabelListSEGSTATE λ segst →
  getSegm tr segm segst :: []

```

```

LabelListLabelTrainsSetSegm : List LabelTrains
LabelListLabelTrainsSetSegm = LUnion LabelListTRAIN      λ tr →
  LUnion LabelListSEGMENT λ segm →
  LUnion LabelListSEGSTATE λ segst →
  setSegm tr segm segst :: []

```

```

LabelListLabelTrainsSetSig : List LabelTrains
LabelListLabelTrainsSetSig = LUnion LabelListTRAIN      λ tr →
                             LUnion LabelListSIGNAL      λ sig →
                             LUnion LabelListASPECT      λ asp →
                             setSig tr sig asp :: []

```

```

LabelListLabelTrainsSetSigs : List LabelTrains
LabelListLabelTrainsSetSigs = LUnion LabelListSIGNAL      λ sig →
                             LUnion LabelListASPECT      λ asp →
                             setSigs sig asp :: []

```

```

LabelListLabelTrains : List LabelTrains
LabelListLabelTrains = LabelListLabelTrainsGetSegm
                      ++! LabelListLabelTrainsSetSegm
                      ++! LabelListLabelTrainsSetSig
                      ++! LabelListLabelTrainsSetSigs

```

```

labelTrains : LUniv
LUniv.Labelf labelTrains = LabelTrains
LUniv._==If_ labelTrains = _==LabelTrains_
LUniv.refl==If labelTrains {l} = refl==labelTrains l
LUniv.showLabelf labelTrains = showLabelTrains
LUniv.LabelListf labelTrains = LabelListLabelTrains
LUniv.sym==If labelTrains {l} {l'} = sym==labelTrains l l'
LUniv.transf labelTrains {l} {l'} = transflabelTrains l l'

```

```

|□|tr : {i : Size} {c : Choice} {A : Set} → List A → (A → Process i {labelTrains} c) → Proc
|□|tr l f = |□| {lu = labelTrains} l f

```

```

--@BEGIN@SIGCTL

SIGCTL : {i : Size} (sig : SIGNAL) → Process $\infty$  i {labelTrains}  $\emptyset'$ 
forcep (SIGCTL sig) = | $\square$ | {lu = labelTrains}      LabelListTRAIN   $\lambda$  tr →
                    | $\square$ | {lu = labelTrains}      LabelListASPECT  $\lambda$  asp →
                    lab (setSig tr sig asp) → SIGCTL sig
Str $\infty$  (SIGCTL sig) = "SIGCTL" ++s showSIGNAL sig

--@END

setSTOP : Choice
setSTOP = ((fin zero)  $\uplus'$  (fin zero))

 $\square$ toStringSimple : String → String → String
 $\square$ toStringSimple str str' = str ++s "  $\square$  " ++s str'

 $\square$ fmapNameSimple : String → String
 $\square$ fmapNameSimple str = "fmap (" ++s str ++s ")"

--@BEGIN@SEGCTL

mutual
  SEGCTL1 : {i : Size} (seg : SEGMENT) (segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTL1 seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
                                lab (getSegm tr seg segstate)
                                → SEGCTL seg segstate
  Str $\infty$  (SEGCTL1 seg segstate) = "SEGCTL1" ++s showSEGMENT seg
                                ++s showSEGSTATE segstate

  SEGCTL2 : {i : Size} (seg : SEGMENT) (segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTL2 seg segstate) = | $\square$ |tr LabelListTRAIN       $\lambda$  tr →
                                | $\square$ |tr LabelListSEGSTATE  $\lambda$  newsegstate →

```

```

                                lab (setSegm tr seg newsegstate)
                                → SEGCTL seg newsegstate
Str∞ (SEGCTL2 seg segstate) = "SEGCTL2" ++s showSEGMENT seg
                                ++s showSEGSTATE segstate

SEGCTL : {i : Size} (seg : SEGMENT) (segstate : SEGSTATE)
        → Process∞ i {labelTrains} ∅'
forcep (SEGCTL seg segstate) = fmap {labelTrains} ∅⊔∅→∅
    ( forcep (SEGCTL1 seg segstate) □wNam
      forcep (SEGCTL2 seg segstate)
      Using □toStringSimple , □fmapNameSimple ,
              □fmapNameSimple)
Str∞ (SEGCTL seg segstate) = "SEGCTL" ++s showSEGMENT seg
                                ++s showSEGSTATE segstate

--@END

mutual
--@BEGIN@TRAINENTER

TRAINENTER : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
            → Process∞ i {labelTrains} ∅'
forcep (TRAINENTER tr seg sig) = lab (getSegm tr seg free)
    →pp ((lab (setSig tr sig green)
              →pp (lab (setSegm tr seg blocked)
                        → TRAINLEAVE tr seg sig)))
Str∞ (TRAINENTER tr seg sig) = "TRAINENTER" ++s showTRAIN tr ++s
                                showSEGMENT seg ++s showSIGNAL sig

--@END

--@BEGIN@TRAINLEAVE

TRAINLEAVE : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
            → Process∞ i {labelTrains} ∅'
forcep (TRAINLEAVE tr seg sig) = lab (setSegm tr seg free)
    →pp (lab (setSig tr sig red)
          → TRAINENTER tr seg sig)
Str∞ (TRAINLEAVE tr seg sig) = "TRAINLEAVE" ++s showTRAIN tr ++s
                                showSEGMENT seg ++s showSIGNAL sig

```

```
--@END
```

```
|||toStringSimple : String → String → String
|||toStringSimple str str' = str ++s " ||| " ++s str'
```

```
fmapNameSimple : ∀{c} → ChoiceSet c → String → String
fmapNameSimple {c} a str = "fmap (" ++s str ++s ")"
```

```
--@BEGIN@SYSTEMpartone
SYSTEMp1 : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅' ×' ∅')
SYSTEMp1 = (SIGCTL sig1 |||wNam∞ SIGCTL sig2
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple)
  |||wNam∞ SEGCTL seg1 free
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple
--@END
```

```
-- ***** change last part to SEGCTL to be defined *****
```

```
[|]|toStringSimple : String → String → String
[|]|toStringSimple str str' = str ++s "[|]|" ++s str'
```

```
--@BEGIN@SYSTEMparttwo
```

```
SYSTEMp2 : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅')
SYSTEMp2 = TRAINENTER ta seg1 sig1 |||wNam∞
  TRAINENTER tb seg1 sig2 Using
  |||toStringSimple , fmapNameSimple , fmapNameSimple
```

```
--@END
```

```
--@BEGIN@SYSTEM
```

```
SYSTEM : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
```

```
SYSTEM = SYSTEMp1 [ (λ x → true) ] ||wNam∞ [ (λ x → true) ] SYSTEMp2
      Using [||]toStringSimple , fmapNameSimple , fmapNameSimple
```

```
--@END
```

```
nameHideInSystem : String → String
nameHideInSystem str = "Hide (" ++s str ++s ")"
```

```
nameEmpty : String → String
nameEmpty str = ""
```

```
--@BEGIN@SYSTEMSHIDE
```

```
hideInSystem : Label labelTrains → Bool
hideInSystem (lab (setSig _ _ _)) = false
hideInSystem (lab (setSigs _ _)) = false
hideInSystem l = true
```

```
SYSTEMSHIDE : {i : Size} → Process∞ i {labelTrains}
      (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
```

```
SYSTEMSHIDE = HideWithName∞ nameHideInSystem hideInSystem SYSTEM
```

```
--@END
```

```
nameRenamInSystem : String → String
nameRenamInSystem str = "Renam (" ++s str ++s ")"
```

```
--@BEGIN@SYSTEMSIGONLY
```

```
renamInSystem : Label labelTrains → Label labelTrains
renamInSystem (lab (setSig x x1 x2)) = lab (setSigs x1 x2)
renamInSystem l = l
```

```
SYSTEM-SIGONLY : {i : Size} → Process∞ i {labelTrains}
      (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
```

```
SYSTEM-SIGONLY = RenameWithName∞ nameRenamInSystem
      renamInSystem SYSTEMSHIDE
```

```
--@END
```

```
mainSIGCTL : IOConsole ∞ Unit
mainSIGCTL = myProgrami ∞ false (SIGCTL sig1)
```

```
main'' : NativeIO Unit
main'' = compile SYSTEMSHIDE
```

```
main' : NativeIO Unit
main' = compile SYSTEMp1
```

```
BAD SIGNALSstr1 : String → String → String
BAD SIGNALSstr1 s s' = "BAD SIGNALS"
```

```
BAD SIGNALSstr2 : String → String
BAD SIGNALSstr2 s = "BAD SIGNALS"
```

```
BAD SIGNALSstr3 : String → String
BAD SIGNALSstr3 s = "BAD SIGNALS"
```

```
efq ∅ ⊔ ∅ : ∀ {X : Set} → ChoiceSet (∅' ⊔ ∅') → X
efq ∅ ⊔ ∅ (inj1 ())
efq ∅ ⊔ ∅ (inj2 ())
```

```
BAD SIGNALS : Process ∞ ∞ {labelTrains} (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
BAD SIGNALS = fmap ∞ efq ∅ ⊔ ∅
  ((lab (setSigs sig1 green) →p ∞ (lab (setSigs sig2 green)
    → STOP ∞ (fin zero)))
  □ wNam ∞ ∞
  (lab (setSigs sig2 green) →p ∞ (lab (setSigs sig1 green)
    → STOP ∞ (fin zero))))
Using BAD SIGNALSstr1 , BAD SIGNALSstr2 , BAD SIGNALSstr3)
```

```
main : NativeIO Unit
main = compile SYSTEM-SIGONLY
```

```
--@PREFIX@trainExampleCorrected

module trainExampleCorrected where
--module trainExample where

open import Data.Bool hiding (==_)
open import Data.Sum

open import Data.Bool.Base renaming (T to T') hiding (==_)
open import libBool
open import libList
open import Data.String
open import Data.String renaming (==_ to ==strb_; ++_ to ++s_)
open import Data.List renaming (++_ to ++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (==_)
open import parallelSimple
open import interleave
open import hidingOperator
open import renamingOperator
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEMSHIDE ; SYSTEM-SIGONLY ; main ; TRAINEN
```

```

--@BEGIN@SEGCTL
mutual
  SEGCTLa : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTLa seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
    lab (getSegm tr seg segstate)
    → SEGCTL1 seg segstate
  Str $\infty$  (SEGCTLa seg segstate) = "SEGCTLa" ++s showSEGMENT seg
    ++s showSEGSTATE segstate

  SEGCTLb : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTLb seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
    | $\square$ |tr LabelListSEGSTATE  $\lambda$  newsegstate →
    lab (setSegm tr seg newsegstate)
    → SEGCTL seg newsegstate
  Str $\infty$  (SEGCTLb seg segstate) = "SEGCTLb" ++s showSEGMENT seg
    ++s showSEGSTATE segstate

  SEGCTL1 : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTL1 seg segstate) = | $\square$ |tr LabelListTRAIN  $\lambda$  tr →
    | $\square$ |tr LabelListSEGSTATE  $\lambda$  newsegstate →
    lab (setSegm tr seg newsegstate)
    → SEGCTL seg newsegstate
  Str $\infty$  (SEGCTL1 seg segstate) = "SEGCTL1" ++s showSEGMENT seg
    ++s showSEGSTATE segstate

  SEGCTL : {i : Size}(seg : SEGMENT)(segstate : SEGSTATE)
    → Process $\infty$  i {labelTrains}  $\emptyset'$ 
  forcep (SEGCTL seg segstate) = fmap  $\emptyset \uplus \emptyset \rightarrow \emptyset$ 
    ( forcep (SEGCTLa seg segstate)  $\square$ wNam
      forcep (SEGCTLb seg segstate)
      Using  $\square$ toStringSimple ,  $\square$ fmapNameSimple ,
         $\square$ fmapNameSimple )
  Str $\infty$  (SEGCTL seg segstate) = "SEGCTL" ++s showSEGMENT seg
    ++s showSEGSTATE segstate

--@END

```

```

mutual
--@BEGIN@TRAINENTER

TRAINENTER : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
  → Process∞ i {labelTrains} ()'
forcep (TRAINENTER tr seg sig) = lab (getSegm tr seg free)
  → pp ((lab (setSig tr sig green)
  → pp (lab (setSegm tr seg blocked)
  → TRAINLEAVE tr seg sig)))
Str∞ (TRAINENTER tr seg sig) = "TRAINENTER" ++s showTRAIN tr ++s
  showSEGMENT seg ++s showSIGNAL sig

--@END

--@BEGIN@TRAINLEAVE

TRAINLEAVE : {i : Size} (tr : TRAIN) (seg : SEGMENT) (sig : SIGNAL)
  → Process∞ i {labelTrains} ()'
forcep (TRAINLEAVE tr seg sig) = lab (setSig tr sig red)
  → pp (lab (setSegm tr seg free)
  → TRAINENTER tr seg sig)
Str∞ (TRAINLEAVE tr seg sig) = "TRAINLEAVE" ++s showTRAIN tr ++s
  showSEGMENT seg ++s showSIGNAL sig

--@END

--@BEGIN@SYSTEMpartone
SYSTEMp1 : {i : Size} → Process∞ i {labelTrains} (()' ×' ()' ×' ()')
SYSTEMp1 = (SIGCTL sig1 |||wNam∞ SIGCTL sig2
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple)
  |||wNam∞ SEGCTL seg1 free
  Using |||toStringSimple , fmapNameSimple , fmapNameSimple
--@END

--@BEGIN@SYSTEMparttwo

```

```

SYSTEMp2 : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅')
SYSTEMp2 = TRAINER ta seg1 sig1 |||wNam∞
          TRAINER tb seg1 sig2 Using
          |||toStringSimple , fmapNameSimple , fmapNameSimple

--@END

--@BEGIN@SYSTEM

SYSTEM : {i : Size} → Process∞ i {labelTrains} (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
SYSTEM = SYSTEMp1 [ (λ x → true) ] |||wNam∞ [ (λ x → true) ] SYSTEMp2
          Using [|||toStringSimple , fmapNameSimple , fmapNameSimple

--@END

--@BEGIN@SYSTEMSHIDE
SYSTEMSHIDE : {i : Size} → Process∞ i {labelTrains}
              (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
SYSTEMSHIDE = HideWithName∞ nameHideInSystem hideInSystem SYSTEM
--@END

--@BEGIN@SYSTEMSIGONLY
SYSTEM-SIGONLY : {i : Size} → Process∞ i {labelTrains}
                (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))
SYSTEM-SIGONLY = RenameWithName∞ nameRenamInSystem
                renameInSystem SYSTEMSHIDE
--@END

main : NativeIO Unit
main = compile SYSTEM-SIGONLY

```

A.118 trainExampleCorrectedOptimized.agda

```
--@PREFIX@trainExampleCorrectedOptimized
```

```

module trainExampleCorrectedOptimized where

open import Data.Bool hiding (==_)
open import Data.Sum
open import Data.Bool.Base renaming (T to T') hiding (==_)
open import libBool
open import libList
open import Data.String
open import Data.String renaming (==_ to ==strb_; ++_ to ++s_)
open import Data.List renaming (++_ to ++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (==_)
open import parallelSimple
open import interleave
open import hidingOperator
open import renamingOperator
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction hiding (test)
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEMp2 ;
SYSTEMSHIDE ; SYSTEM-SIGONLY ; main ; TRAINENTER)
open import trainExampleCorrected hiding (main)
open import process2OptimizedProcess

```

```
OPTIMIZED-SYSTEM : Process $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
OPTIMIZED-SYSTEM = optimizedProcess $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ ) SYSTEM-SIGONLY
```

```
main : NativeIO Unit
main = compile OPTIMIZED-SYSTEM
```

```
test : Process  $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
test = forcep OPTIMIZED-SYSTEM { $\infty$ }
```

```
test1 : Choice
test1 = Ep test
```

```
test2 : Choice
test2 = lp test
```

```
test3 : Choice
test3 = Tp test
```

A.119 trainExampleCorrectedOptimized2.agda

```
--@PREFIX@trainExampleCorrectedOptimizedTwo
```

```
module trainExampleCorrectedOptimized2 where
```

```
open import Data.Bool hiding ( $\_==\_$ )
open import Data.Sum
open import Data.Bool.Base renaming (T to T') hiding ( $\_==\_$ )
open import libBool
open import libList
open import Data.String
open import Data.String renaming ( $\_==\_$  to  $\_==\text{strb}\_$ ;  $\_++\_$  to  $\_++\text{s}\_$ )
open import Data.List renaming ( $\_++\_$  to  $\_++\text{l}\_$ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
```

```

open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (=?=)
open import parallelSimple
open import interleave
open import hidingOperator
open import renamingOperator
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction hiding (test)
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEMp2
                                SYSTEMSHIDE ; SYSTEM-SIGONLY ; main ; TRAINENTER)

open import trainExampleCorrected hiding (main)
open import choiceSetUOptimized2
open import process2OptimizedProcess2

```

```

OPTIMIZED-SYSTEM : Process $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
OPTIMIZED-SYSTEM = optimizedProcess $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ ) SYSTEM-SIGONLY

```

```

main : NativeIO Unit
main = compile OPTIMIZED-SYSTEM

```

```

test : Process  $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
test = forcep OPTIMIZED-SYSTEM { $\infty$ }

```

```

test1 : Choice
test1 = Ep test

```

```

test2 : Choice
test2 = lp test

```

test3 : Choice
test3 = Tp test

A.120 trainExampleCorrectedOptimized3.agda

```
--@PREFIX@trainExampleCorrectedOptimizedThree
```

```
module trainExampleCorrectedOptimized3 where
```

```
open import Data.Bool hiding (_==_)
open import Data.Sum
open import Data.Bool.Base renaming (T to T') hiding (_==_)
open import libBool
open import libList
open import Data.String
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List renaming (_++_ to _++l_; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin
open import Data.Nat hiding (_==_)
open import parallelSimple
open import interleave
open import hidingOperator
open import renamingOperator
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction hiding (test)
open import Relation.Nullary.Decidable
open import Data.String
```

```

open import UnitModule
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEMp2
                                SYSTEMSHIDE ; SYSTEM-SIGONLY ; main ; TRAINENTER
open import trainExampleCorrected hiding (main)
open import choiceSetUOptimized3
open import process2OptimizedProcess3

```

```

OPTIMIZED-SYSTEM : Process $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
OPTIMIZED-SYSTEM = optimizedProcess $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ ) SYSTEM-SIGONLY

```

```

main : NativeIO Unit
main = compile OPTIMIZED-SYSTEM

```

```

test : Process  $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
test = forcep OPTIMIZED-SYSTEM { $\infty$ }

```

```

test1 : Choice
test1 = Ep test

```

```

test2 : Choice
test2 = lp test

```

```

test3 : Choice
test3 = Tp test

```

A.121 trainExampleCorrectedOptimized3ProofNonRef1.agda

```
--@PREFIX@trainExampleCorrectedOptimizedThreeProofNonRefinement
```

```

module trainExampleCorrectedOptimized3ProofNonRefinement1 where
--module trainExample where

```

```

open import Data.Bool hiding (_=?)
open import Data.Sum
open import Data.Unit
open import Relation.Binary.PropositionalEquality

```

```

open import Data.Bool.Base renaming (T to T') hiding (=?_)
open import libBool
open import dataAuxFunction renaming (efq to efq1)
open import process
open import libList
open import Data.String
open import Data.Maybe
open import Data.String renaming (==_ to ==strb_; ++_ to ++s_)
open import Data.List renaming (++_ to ++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (=?_)
open import parallelSimple
open import interleaved
open import auxData
open import hidingOperator
open import renamingOperator
{- needed possibly for compilation-}

open import SizedIO.Base hiding (delay)
open import renamingResult
open import dataAuxFunction hiding (¬)
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import Data.Empty
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEM ; SYSTEM
                                SYSTEM-SIGONLY ; TRAINLEAVE ; TRAINENTER ; main)
open import trainExampleCorrected
open import TraceWithoutSize
open import RefWithoutSize

```

```

-- open import dataAuxFunction
open import choiceSetUOptimized3
open import process2OptimizedProcess3
open import trainExampleCorrectedOptimized3

-- lemma1notfin0 : (((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0) ×
-- ((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0))
-- × ((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0))
-- × (Fin 0 × Fin 0) (((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0) ×
-- ((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0))
-- × ((Fin 0 ⊔ Fin 0) ⊔ Fin 0 ⊔ Fin 0))
-- × (Fin 0 × Fin 0) → ⊥
-- lemma1notfin0 x = ?

strAppend : String → String → String
strAppend = primStringAppend

f0 : Choice
f0 = fin 0

f1 : Choice
f1 = fin 1

badTraceLabels : List (Label labelTrains)
badTraceLabels = lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: []

badTraceBadSignal : Tr∞ {labelTrains} badTraceLabels nothing BADSIGNALS
badTraceBadSignal = tnode (extc (lab (setSigs sig2 green) :: []) nothing (inj₁ zero)
  (tnode (extc [] nothing zero
    (tnode empty))))))

noTraceBadSignal : (m : Maybe (ChoiceSet (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))))
  (l : List (Label labelTrains))
  (tr : Tr∞ {labelTrains} l m OPTIMIZED-SYSTEM)
  (mm' : m ≡ nothing)
  (ll' : l ≡ badTraceLabels)
  → ⊥
noTraceBadSignal .nothing .[] (tnode empty) mm' ()
noTraceBadSignal m
  .(renameInSystem (Lab (((fmap+ c⊔'c->c (fmap+ c⊔'c->c (process+ f1 (λ _ → lab (setSigs
    efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1

```

```

(λ _ → lab (setSig ta sig1 green)) (λ _ → SIGCTL sig1) f0
efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red)) (λ _ → SIGCTL sig1) f0
efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1 (λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)") |||wNam++
fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1 (λ _ → lab (setSig ta sig2 red)) (λ _ → SIGCTL sig2)
efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1 (λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)") □+ fmap+ c⊕'c->c
(process+ f1 (λ _ → lab (setSig tb sig2 red)) (λ _ → SIGCTL sig2)
f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green)) (λ _ → SIGCTL sig2) f0 efq1 f0
efq1 "(tbsig2green -> SIGCTLsig2)") Using (λ str str' → strAppend str (strAppend " ||| "
(λ a str → strAppend "fmap (" (strAppend str "))) , (λ a str → strAppend "fmap (" (strAppend
|||wNam++ fmap+ ∅⊕∅->∅ (fmap+ c⊕'c->c (process+ f1 (λ _ → lab (getSegm ta seg1 free)) (λ _
efq1 f0 efq1 "(taseg1free -> SEGCTL1seg1free)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 free)) (λ _ → SEGCTL1 seg1 free) f0 efq1 f0 efq1
"(tbseg1free -> SEGCTL1seg1free)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked)) (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
"(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1 (λ _ → lab (setS
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked)) (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
"(tbseg1blocked -> SEGCTLseg1blocked)") Using (λ str str' → strAppend str (strAppend " □
(λ str → strAppend "fmap (" (strAppend str "))) , (λ str → strAppend "fmap (" (strAppend str
Using (λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) , (λ a str → strAppend "fmap (" (strAppend
[ (λ x → true) ] |||wNam+[ (λ x → true) ] process+ f1 (λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1 (λ _ → lab (setSig ta sig1 green)) (λ _ → delay (node
(process+ f1 (λ _ → lab (setSegm ta seg1 blocked)) (λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0
efq1 "(taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0
efq1 "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0
efq1 "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" |||
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1 "(tbseg1blocked -> TRAINLEAVetbseg1sig2)"
efq1 "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))) f0 efq1 f0
efq1 "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))" Usin
(λ str str' → strAppend str (strAppend " ||| " str')) ,

```

```

(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) Using
(λ str str' → strAppend str (strAppend "[]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (projSubset (nth [] _))) :: l₁) (tnode (
noTraceBadSignal .nothing .[] (tnode (intc .[] .nothing zero (tnode empty))) mm' ()
noTraceBadSignal .nothing .(lab (setSigs sig1 green) :: [])
  (tnode (intc .(lab (setSigs sig1 green) :: []) .nothing zero
    (tnode (extc .[] .nothing zero (tnode empty))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig1 green) :: renameInSystem
  (Lab (((fmap+ c⊕'c->c (fmap+ inj₁ (fmap+ c⊕'c->c (fmap+ inj₂
    (fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
      (λ _ → lab (setSig ta sig1 red))
      (λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁ "(tasig1red -> SIGCTLsig1)" □+ process+ f1
      (λ _ → lab (setSig ta sig1 green))
      (λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁ "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c
      (λ _ → lab (setSig tb sig1 red))
      (λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁ "(tb sig1red -> SIGCTLsig1)" □+ process+ f1
      (λ _ → lab (setSig tb sig1 green))
      (λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁ "(tb sig1green -> SIGCTLsig1)" ))))) ||| wNam++
    fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
      (λ _ → lab (setSig ta sig2 red))
      (λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁ "(tasig2red -> SIGCTLsig2)" □+ process+ f1
      (λ _ → lab (setSig ta sig2 green))
      (λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁ "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c
      (λ _ → lab (setSig tb sig2 red))
      (λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁ "(tb sig2red -> SIGCTLsig2)" □+ process+ f1
      (λ _ → lab (setSig tb sig2 green))
      (λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁ "(tb sig2green -> SIGCTLsig2)" )) Using
    (λ str str' → strAppend str (strAppend " ||| " str')) ,
    (λ a str → strAppend "fmap (" (strAppend str ")") ,
    (λ a str → strAppend "fmap (" (strAppend str ")")) ||| wNam++ fmap+ ∅⊕∅->∅ (fmapW
    (λ str → strAppend "fmap (" (strAppend str ")") ) inj₁ (fmap+ c⊕'c->c (fmap+ inj₁ (fmap
    (λ _ → lab (setSegm ta seg1 free))
    (λ _ → SEGCTL seg1 free) f0 efq₁ f0 efq₁ "(taseg1free -> SEGCTLseg1free)" □+ proce
    (λ _ → lab (setSegm ta seg1 blocked))
    (λ _ → SEGCTL seg1 blocked) f0 efq₁ f0 efq₁ "(taseg1blocked -> SEGCTLseg1blocked)"
    (λ _ → lab (setSegm tb seg1 free))
    (λ _ → SEGCTL seg1 free) f0 efq₁ f0 efq₁ "(tbseg1free -> SEGCTLseg1free)" □+ proce
    (λ _ → lab (setSegm tb seg1 blocked))
    (λ _ → SEGCTL seg1 blocked) f0 efq₁ f0 efq₁ "(tbseg1blocked -> SEGCTLseg1blocked)"

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1 "(taseg1blocked -> TRAINLEAVEtaseg1sig1)"
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
"(tbseg1blocked -> TRAINLEAVEtbseg1sig2)))) f0 efq1 f0 efq1
"(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))) f0 efq1 f0 efq1
"(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l)
(tnode (intc .(lab (setSigs sig1 green) :: renameInSystem
(Lab (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2
(fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)))))) ||wNam++ fmap+
c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))

```

```

(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ proce
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)
(process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ proce
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1 "(taseg1blocked -> TRAINLEAVetaseg
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
"(tbseg1blocked -> TRAINLEAVetbseg1sig2)))) f0 efq1 f0 efq1
"(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))) f0 efq1 f0 efq1
"(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))
(projSubset (nth [] _)) :: l) .m zero
(tnode (extc .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fm
(fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1

```

```

(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)" )))) |||wNam++ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)" ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName+ f1
(λ str → strAppend "fmap (" (strAppend str " "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)" )))) U
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) [
(λ x → true) ] |||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))

```

```

(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1 "(tbseg1blocked -> TRAINLEAVEtbseg
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))" f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))"
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l) .m zero (
noTraceBadSignal .nothing .(lab (setSigs sig1 green) :: [])
  (tnode (intc .(lab (setSigs sig1 green) :: []) .nothing zero
    (tnode (extc [] .nothing zero
      (tnode (intc [] .nothing zero (tnode empty)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig1 green) :: lab (setSigs sig1 red) :: l)
  (tnode (intc .(lab (setSigs sig1 green) :: lab (setSigs sig1 red) :: l) .m zero
    (tnode (extc .(lab (setSigs sig1 red) :: l) .m zero
      (tnode (intc .(lab (setSigs sig1 red) :: l) .m zero
        (tnode (extc l .m zero tr)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig1 green)
  :: renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2
    (fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)" )))) ||wNam++ f
  (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)" )) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW

```

```

(λ str → strAppend "fmap (" (strAppend str ")") ) inj1 (fmap+ c⊕'c->c (fmap+ inj1
    (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ ∅⊕∅->∅ (fmap+ c⊕'c->c
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTL1seg1blocked)" □+
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTL1seg1blocked)") □w
    fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)") □+ fr
    (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)")) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str ")") ) ,
(λ str → strAppend "fmap (" (strAppend str ")") ) ) ) ) ) ) ) ) ) ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ) [
(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1 "(taseg1free -> TRAINENTERtaseg1sig1)"))
    "(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" ||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
    "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)")) f0 efq1 f0 efq1
    "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)")) f0 efq1 f0 efq1
    "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) Using

```

```

(λ str str' → strAppend str (strAppend "[]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] -)) :: l)
  (tnode (intc .(lab (setSigs sig1 green) ::
    renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2
      (fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
        (λ _ → lab (setSig ta sig1 red))
        (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" ⊞+
          process+ f1
        (λ _ → lab (setSig ta sig1 green))
        (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" ⊞+ fmap+ c⊕'c->c
        (λ _ → lab (setSig tb sig1 red))
        (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" ⊞+ process+ f1
        (λ _ → lab (setSig tb sig1 green))
        (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)")))))) ||wNam++ f
        (λ _ → lab (setSig ta sig2 red))
        (λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" ⊞+ process+ f1
        (λ _ → lab (setSig ta sig2 green))
        (λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" ⊞+ fmap+ c⊕'c->c
        (λ _ → lab (setSig tb sig2 red))
        (λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" ⊞+ process+ f1
        (λ _ → lab (setSig tb sig2 green))
        (λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)")) Using
      (λ str str' → strAppend str (strAppend " ||| " str')) ,
      (λ a str → strAppend "fmap (" (strAppend str "))) ,
      (λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ⊕⊕->⊕ (fmapW
      (λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+
        (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ ⊕⊕->⊕ (fmap+ c⊕'c->c (process+
          (λ _ → lab (getSegm ta seg1 blocked))
          (λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTL1seg1blocked)
          (λ _ → lab (getSegm tb seg1 blocked))
          (λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTL1seg1blocked)
          (λ _ → lab (setSegm ta seg1 free))
          (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" ⊞+ proce
          (λ _ → lab (setSegm ta seg1 blocked))
          (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)
          (λ _ → lab (setSegm tb seg1 free))
          (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" ⊞+ proce
          (λ _ → lab (setSegm tb seg1 blocked))
          (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)
        (λ str str' → strAppend str (strAppend " □ " str')) ,

```

○

```

(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1
    (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ 0⊕0->0 (fmap+ c⊕
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTL1seg1blocke
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTL1seg1blocke
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ proce
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ proce
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]|||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1
    "(taseg1free -> TRAINENTERtaseg1sig1)") f0 efq1 f0 efq1
    "(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" |||wNam++ process+
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
    "(tbseg1blocked -> TRAINLEAVETbseg1sig2)") f0 efq1 f0 efq1

```

```

      "(tbseg1green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))) f0 efq1 f0 efq1
      "(tbseg1free -> (tbseg1green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l) .m zero
  (tnode (intc .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbseg1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbseg1green -> SIGCTLsig1)")))))) |||wNam++
  fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbseg2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbseg2green -> SIGCTLsig2)")) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName-
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTL1seg1blocked)" □+
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTL1seg1blocked)" □w
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)" □+ fr
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))

```

```

(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1 "(taseg1free -> TRAINENTERtaseg1s
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1 "(tbseg1blocked -> TRAINLEAVEtbseg
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend " [||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l) .m zero (
noTraceBadSignal m .(lab (setSigs sig1 green) :: l1)
  (tnode (intc .(lab (setSigs sig1 green) :: l1) .m zero
  (tnode (extc l1 .m zero
  (tnode (intc .l1 .m zero
  (tnode (intc .l1 .m () tr))))))))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fma
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig tb sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)))))) |||wNam++ f

```

```

(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1)
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName+ f1)
(λ str → strAppend "fmap (" (strAppend str "))) inj1
  (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2
    (fmap+ ∅⊕∅->∅ (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTL1seg1blocked)" □w
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTLseg1blocked)" □+ fr
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked)") Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1free -> TRAINENTERtaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" |||wNam++ process+ f1

```

```

(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)")))) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)")))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)"))))
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) (nth [] _)) .(lab (setSigs sig1 green)
  (tnode (intc .(lab (setSigs sig1 green) :: [])) .(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1 "(tbsig1green -> SIGCTLsig1)")))))) ||wNam++ f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1 "(tbsig2green -> SIGCTLsig2)")) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str " ")) inj1
  (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c
  (fmap+ inj2 (fmap+ ∅⊕∅->∅ (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1 "(taseg1blocked -> SEGCTL1seg1blocked
(λ _ → lab (getSegm tb seg1 blocked))

```

```

(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proc
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1 "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1 "(tbseg1blocked -> SEGCTLseg1blocked))) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1free -> TRAINENTERtaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVETbseg1sig2)))) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVETbseg1sig2)))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVETbseg1sig2))))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] _)) zero
  (tnode (extc .[]).(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c

```

```

(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" )))) ||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (p
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTL1seg1blocked)" ) □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))

```

```

(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked))" Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1free -> TRAINENTERtaseg1sig1))" f0 efq1 f0 efq1
  "(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2))" f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))" f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] -)) zero
  (tnode (intc .[] .(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1

```

```

(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" )))) ||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (p
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" ) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,

```

```

(λ str → strAppend "fmap (" (strAppend str ")") ,
(λ str → strAppend "fmap (" (strAppend str ")") ) ) ) ) ) ) ) ) ) ) ) ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ) [
(λ x → true) ] ||wNam+
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → TRAINENTER ta seg1 sig1) f0 efq1 f0 efq1
"(taseg1free -> TRAINENTERtaseg1sig1)" ) ) ) f0 efq1 f0 efq1
"(tasig1red -> (taseg1free -> TRAINENTERtaseg1sig1))" ||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
"(tbseg1blocked -> TRAINLEAVETbseg1sig2)" ) ) ) f0 efq1 f0 efq1
"(tbsig2green -> (tbseg1blocked -> TRAINLEAVETbseg1sig2))" ) ) ) f0 efq1 f0 efq1
"(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVETbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ) (nth [] _)) zero
(tnode (terc ()))))) mm' ll'
noTraceBadSignal m .(lab (setSigs sig1 green) :: l1)
(tnode (intc .(lab (setSigs sig1 green) :: l1) .m zero
(tnode (extc l1 .m zero
(tnode (intc .l1 .m (suc ()) tr)))))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tasig1green -> SIGCTLsig1)" ) □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))

```

```

(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" )))) ||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (p
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" ||wNam++ process+ f1

```



```

(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
    "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)"))) f0 efq1 f0 efq1
    "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))"))) f0 efq1 f0 efq1
    "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) Using
(λ str str' → strAppend str (strAppend "[| |]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) (nth [] t))) .(lab (setSigs sig1 green) :: [])
    (tnode (intc .(lab (setSigs sig1 green) :: []) .(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)")))))) |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,

```

```

(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) ||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str ")") ) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" ) ) ) ) ) Using
(λ str str' → strAppend str (strAppend " ||| " str') ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) [
(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)" ||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" ) ) ) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" ) ) ) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" )
(λ str str' → strAppend str (strAppend " ||| " str') ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) Using
(λ str str' → strAppend str (strAppend "[|]" str') ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) (nth [] t)) zero
  (tnode (extc .[]).(just (PT (((fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fn
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1

```

```

    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig tb sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)" )))) ||| wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+
  (λ _ → lab (setSig ta sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig ta sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig tb sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)" )) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) ||| wNam++ fmap+ ∅⊕∅->∅ (fmapWithName
  (λ str → strAppend "fmap (" (strAppend str " "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->c
  (λ _ → lab (setSegm ta seg1 free))
  (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
  (λ _ → lab (setSegm ta seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSegm tb seg1 free))
  (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) [

```

```

(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" ||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)")))) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)")))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))) (nth [] t)) zero
  (tnode (terc t)))))) mm' ()
noTraceBadSignal m .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" ⊞+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" ⊞+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" ⊞+ process+ f1
  (λ _ → lab (setSig tb sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)")) ||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proce
  (λ _ → lab (setSig ta sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" ⊞+ process+ f1
  (λ _ → lab (setSig ta sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" ⊞+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig2 red))

```

```

(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName-
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap+ c⊕'c->
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)))) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))))" Using

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l₁)
  (tnode (intc .(renamelnSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq₁ f0 efq₁
  "(tbsig1green -> SIGCTLsig1)" |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proce
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq₁ f0 efq₁
  "(tbsig2green -> SIGCTLsig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj₁ (fmap+ c⊕'c->c (fmap+ inj₁ (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq₁ f0 efq₁
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq₁ f0 efq₁
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1

```

```

(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)" )) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" )) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" )) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l1) .m zero
  (tnode (extc l1 .m (suc ()) tr))) mm' ll'
noTraceBadSignal m l
  (tnode (intc .l .m zero
  (tnode (intc .l .m () tr))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊔'c->c (fmap+ c⊔'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))

```



```

(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proce
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ] |||wNam+[
(λ x → true) ] process+ f1

```



```

(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)"))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" |||wNam++ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)"))) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" ))) f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) (nth [] t))) .[]
  (tnode (intc [] .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" ⊞+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" ⊞+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" ⊞+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" ))) |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" ⊞+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" ⊞+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))

```

```

(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj1 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]|||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)") f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))" |||wNam++ proce
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtaseg1sig2)") f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtaseg1sig2)") f0 efq1 f0 efq1
  "(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtaseg1sig2)))

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) Using
(λ str str' → strAppend str (strAppend "[| |]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (nth [] t))) zero
  (tnode (terc t))) mm' ()
noTraceBadSignal .nothing .[]
  (tnode (intc .[] .nothing (suc zero)
  (tnode empty))) mm' ()
noTraceBadSignal .nothing .(lab (setSigs sig2 green) :: [])
  (tnode (intc .(lab (setSigs sig2 green) :: []) .nothing (suc zero)
  (tnode (extc .[] .nothing zero
  (tnode empty)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig2 green) :: renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" □+ ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" ))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,

```

```

(λ a str → strAppend "fmap (" (strAppend str ")")) |||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str ")")) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) [
(λ x → true) ] |||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))))
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (projSubset (nth [] _)) :: l)
  (tnode (intc .(lab (setSigs sig2 green) :: renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1

```

```

(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName-
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]|||wNam+[

```

```

(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" ))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))" ))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) (projSubset (nth [] _)) :: l) .m (suc z
  (tnode (extc .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" ⊞+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" ⊞+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" ⊞+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" ))) ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" ⊞+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" ⊞+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" ⊞+ process+ f1

```

```

(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapWithName
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ] |||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" )))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)" )))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))" |||wNam
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVEtbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,

```

```

    (λ a str → strAppend "fmap (" (strAppend str ")")) (projSubset (nth [] _)) :: l) .m zero
    (tnode (extc l .m () tr)))))) mm' ll'
noTraceBadSignal .nothing .(lab (setSigs sig2 green) :: [])
  (tnode (intc .(lab (setSigs sig2 green) :: []) .nothing (suc zero)
    (tnode (extc .[] .nothing zero
      (tnode (intc .[] .nothing zero
        (tnode empty)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig2 green) :: lab (setSigs sig2 red) :: l)
  (tnode (intc .(lab (setSigs sig2 green) :: lab (setSigs sig2 red) :: l) .m (suc zero)
    (tnode (extc .(lab (setSigs sig2 red) :: l) .m zero
      (tnode (intc .(lab (setSigs sig2 red) :: l) .m zero
        (tnode (extc l .m zero tr)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig2 green) :: renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig tb sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)")) |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕
  (λ _ → lab (setSig ta sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig ta sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig tb sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)")))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str ")")) ,
  (λ a str → strAppend "fmap (" (strAppend str ")")) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
  (λ str → strAppend "fmap (" (strAppend str ")")) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+

```

```

(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proc
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)") Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1")))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))")) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))" |||wNam
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2")))) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l)
  (tnode (intc .(lab (setSigs sig2 green) :: renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c

```

```

(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked))) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))))" |||wNam
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)))) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2)))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] -))) :: l) .m (suc zero)

```

```

    (tnode (extc .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)") □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)") □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))),
(λ a str → strAppend "fmap (" (strAppend str " ")))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str " "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)") □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)") □+ fmap+ c⊕'c->c (process+ f1

```

```

(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked))" Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str ")")) ,
(λ str → strAppend "fmap (" (strAppend str ")"))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) [
(λ x → true) ]||wNam+
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1))" f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" |||wNam
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2))" f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (projSubset (nth [] _)) :: l) .m zero
  (tnode (intc .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1

```

```

    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)") ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)") Using

```

```

(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str ")") ,
(λ str → strAppend "fmap (" (strAppend str ")") ) ) ) ) ) ) ) ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) [
(λ x → true) ] ||wNam+
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)") ) ) ) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)") ) ) ) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" ||wNam+
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)") ) ) ) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) (projSubset (nth [] _)) :: l) .m zero
  (tnode (extc l .m (suc ()) tr)))))) mm' ll'
noTraceBadSignal m .(lab (setSigs sig2 green) :: l1)
  (tnode (intc .(lab (setSigs sig2 green) :: l1) .m (suc zero)
  (tnode (extc l1 .m zero
  (tnode (intc .l1 .m zero
  (tnode (intc .l1 .m () tr)))))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1

```

```

    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)") Using

```

```

(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str ")") ,
(λ str → strAppend "fmap (" (strAppend str ")") ) ) ) ) ) ) ) ) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) [
(λ x → true) ] ||wNam+
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)" ) ) ) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)" ) ) ) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" ||wNam+
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)" ) ) ) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")") ,
(λ a str → strAppend "fmap (" (strAppend str ")") ) (nth [] t)) .(lab (setSigs sig2 green) :: [])
  (tnode (intc .(lab (setSigs sig2 green) :: []) .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1

```

```

    "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked))) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str "))) ,
(λ str → strAppend "fmap (" (strAppend str "))))))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [

```

```

(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)" ))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))" ))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))" |||wNam
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)" ))) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) ,
(λ a str → strAppend "fmap (" (strAppend str " ")) (nth [] t)) (suc zero)
  (tnode (extc .[] .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" ))) |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1

```

```

    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str " "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (getSegm ta seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 blocked))
(λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c-
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)" )) Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str " "))) ,
(λ str → strAppend "fmap (" (strAppend str " ")))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) [
(λ x → true) ] ||wNam+ [
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1

```

```

(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)" f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" |||wNam
(λ _ → lab (setSig tb sig2 red))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)" f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[ | ]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] t)) zero
  (tnode (intc .[] .just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)" |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1

```

```

    "(tbsig2green -> SIGCTLsig2)" )))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) |||wNam++ fmap+ 0⊕0->0 (fmapW
  (λ str → strAppend "fmap (" (strAppend str " "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
  (λ _ → lab (getSegm ta seg1 blocked))
  (λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTL1seg1blocked)" ⊞+ process+ f1
  (λ _ → lab (getSegm tb seg1 blocked))
  (λ _ → SEGCTL1 seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTL1seg1blocked)" ⊞wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c-
  (λ _ → lab (setSegm ta seg1 free))
  (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(taseg1free -> SEGCTLseg1free)" ⊞+ process+ f1
  (λ _ → lab (setSegm ta seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(taseg1blocked -> SEGCTLseg1blocked)" ⊞+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSegm tb seg1 free))
  (λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
    "(tbseg1free -> SEGCTLseg1free)" ⊞+ process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)" )) Using
  (λ str str' → strAppend str (strAppend " □ " str')) ,
  (λ str → strAppend "fmap (" (strAppend str " "))) ,
  (λ str → strAppend "fmap (" (strAppend str " "))) )))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) ,
  (λ a str → strAppend "fmap (" (strAppend str " "))) [
  (λ x → true) ] |||wNam+[
  (λ x → true) ] process+ f1
  (λ _ → lab (getSegm ta seg1 free))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSegm ta seg1 blocked))
  (λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
    "(taseg1blocked -> TRAINLEAVetaseg1sig1)" )) f0 efq1 f0 efq1
    "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" )) f0 efq1 f0 efq1
    "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))"
  (λ _ → lab (setSig tb sig2 red))

```

```

(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → TRAINENTER tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1free -> TRAINENTERtbseg1sig2)")) f0 efq1 f0 efq1
  "(tbsig2red -> (tbseg1free -> TRAINENTERtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str ")))) (nth [] t)) zero
  (tnode (terc t)))))) mm' ()
noTraceBadSignal m .(lab (setSigs sig2 green) :: l1)
  (tnode (intc .(lab (setSigs sig2 green) :: l1) .m (suc zero)
  (tnode (extc l1 .m zero
  (tnode (intc .l1 .m (suc ()) tr)))))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig tb sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1green -> SIGCTLsig1)")) |||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
  (λ _ → lab (setSig ta sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig ta sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig2 red))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
  (λ _ → lab (setSig tb sig2 green))
  (λ _ → SIGCTL sig2) f0 efq1 f0 efq1
    "(tbsig2green -> SIGCTLsig2)")))))) Using

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]|||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))))
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] t))) .(lab (setSigs sig2 green) :
  (tnode (intc .(lab (setSigs sig2 green) :: [])) .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c
(λ _ → lab (setSig ta sig1 red))

```

```

(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c (f
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,

```

```

(λ a str → strAppend "fmap (" (strAppend str ")")) [
(λ x → true) ] ||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVEtaseg1sig1)")))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))")) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVETbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (nth [] t)) (suc zero)
  (tnode (extc .[]).(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)")) ||wNam++ fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))

```

```

(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapWithName
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ] |||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)" )))) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)" )))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" |||wNam
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using

```

```

(λ str str' → strAppend str (strAppend "[|]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] t))) zero
(tnode (terc t)))))) mm' ()
noTraceBadSignal m .(renameInSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
"(tbsig1green -> SIGCTLsig1)" |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
"(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
"(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
"(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
"(tbsig2green -> SIGCTLsig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
"(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
"(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1

```

```

    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) [
  (λ x → true) ] ||wNam+ [
  (λ x → true) ] process+ f1
  (λ _ → lab (getSegm ta seg1 free))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSegm ta seg1 blocked))
  (λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
    "(taseg1blocked -> TRAINLEAVetaseg1sig1)" )))) f0 efq1 f0 efq1
    "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" )))) f0 efq1 f0 efq1
    "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))" |||wNam
  (λ _ → lab (setSig tb sig2 green))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
    "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" )))) f0 efq1 f0 efq1
    "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) Using
  (λ str str' → strAppend str (strAppend "[|]" str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _)) :: l1)
  (tnode (intc .(renamelnSystem (Lab (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig tb sig1 green))

```

```

(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (proce
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ ∅⊕∅->∅ (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)))))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]|||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1

```

```

      "(taseg1blocked -> TRAINLEAVEtaseg1sig1))))) f0 efq1 f0 efq1
      "(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))))) f0 efq1 f0 efq1
      "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))" |||wNam
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
      "(tbseg1blocked -> TRAINLEAVEtbseg1sig2))))) f0 efq1 f0 efq1
      "(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[| |]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (projSubset (nth [] _))) :: l1 .m (suc zero)
      (tnode (extc l1 .m (suc ()) tr))) mm' ll'
noTraceBadSignal m l
      (tnode (intc .l .m (suc zero)
      (tnode (intc .l .m () tr)))) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
      "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
      "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
      "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
      "(tbsig1green -> SIGCTLsig1))) |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
      "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
      "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
      "(tbsig2red -> SIGCTLsig2)" □+ process+ f1

```

```

(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) |||wNam++ fmap+ 0⊕0->0 (fmapW
(λ str → strAppend "fmap (" (strAppend str "))) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)")))) Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
  "(taseg1blocked -> TRAINLEAVetaseg1sig1)")) f0 efq1 f0 efq1
  "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))) f0 efq1 f0 efq1
  "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
  "(tbseg1blocked -> TRAINLEAVetbseg1sig2)")) f0 efq1 f0 efq1
  "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,

```

```

(λ a str → strAppend "fmap (" (strAppend str ")")) Using
(λ str str' → strAppend str (strAppend "[]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) (nth [] t))) .[]
  (tnode (intc .[] .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig1 red))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) ,
(λ a str → strAppend "fmap (" (strAppend str ")")) |||wNam++ fmap+ ∅⊕∅->∅ (fmapWithName
(λ str → strAppend "fmap (" (strAppend str ")")) inj1 (fmap+ c⊕'c->c (fmap+ inj2 (fmap+ c⊕'c->c
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1

```

```

    "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
    "(tbseg1blocked -> SEGCTLseg1blocked)" )))) Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) [
  (λ x → true) ]||wNam+[
  (λ x → true) ] process+ f1
  (λ _ → lab (getSegm ta seg1 free))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSegm ta seg1 blocked))
  (λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
    "(taseg1blocked -> TRAINLEAVetaseg1sig1)" )))) f0 efq1 f0 efq1
    "(tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1))" )))) f0 efq1 f0 efq1
    "(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVetaseg1sig1)))
  (λ _ → lab (setSig tb sig2 green))
  (λ _ → delay (node (process+ f1
  (λ _ → lab (setSegm tb seg1 blocked))
  (λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
    "(tbseg1blocked -> TRAINLEAVetbseg1sig2)" )))) f0 efq1 f0 efq1
    "(tbsig2green -> (tbseg1blocked -> TRAINLEAVetbseg1sig2))" Using
  (λ str str' → strAppend str (strAppend " ||| " str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) Using
  (λ str str' → strAppend str (strAppend "[||]" str')) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) ,
  (λ a str → strAppend "fmap (" (strAppend str "))) (nth [] t))) (suc zero)
  (tnode (terc t))) mm' ()
noTraceBadSignal m l
  (tnode (intc .l .m (suc (suc ())) tr)) mm' ll'
noTraceBadSignal .(just (PT (((fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig ta sig1 red))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1red -> SIGCTLsig1)" □+ process+ f1
  (λ _ → lab (setSig ta sig1 green))
  (λ _ → SIGCTL sig1) f0 efq1 f0 efq1
    "(tasig1green -> SIGCTLsig1)" □+ fmap+ c⊕'c->c (process+ f1
  (λ _ → lab (setSig tb sig1 red))

```

```

(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1red -> SIGCTLsig1)" □+ process+ f1
(λ _ → lab (setSig tb sig1 green))
(λ _ → SIGCTL sig1) f0 efq1 f0 efq1
  "(tbsig1green -> SIGCTLsig1)") |||wNam++ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig ta sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig ta sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tasig2green -> SIGCTLsig2)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSig tb sig2 red))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2red -> SIGCTLsig2)" □+ process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → SIGCTL sig2) f0 efq1 f0 efq1
  "(tbsig2green -> SIGCTLsig2)") Using
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str " "))) ,
(λ a str → strAppend "fmap (" (strAppend str " ")))) |||wNam++ fmap+ ∅⊕∅->∅ (fmap+ c⊕'c->c
(λ _ → lab (getSegm ta seg1 free))
(λ _ → SEGCTL1 seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTL1seg1free)" □+ process+ f1
(λ _ → lab (getSegm tb seg1 free))
(λ _ → SEGCTL1 seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTL1seg1free)" □wNam+ fmap+ c⊕'c->c (fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm ta seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(taseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(taseg1blocked -> SEGCTLseg1blocked)" □+ fmap+ c⊕'c->c (process+ f1
(λ _ → lab (setSegm tb seg1 free))
(λ _ → SEGCTL seg1 free) f0 efq1 f0 efq1
  "(tbseg1free -> SEGCTLseg1free)" □+ process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → SEGCTL seg1 blocked) f0 efq1 f0 efq1
  "(tbseg1blocked -> SEGCTLseg1blocked)") Using
(λ str str' → strAppend str (strAppend " □ " str')) ,
(λ str → strAppend "fmap (" (strAppend str " "))) ,
(λ str → strAppend "fmap (" (strAppend str " ")))) Using

```

```

(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) [
(λ x → true) ]||wNam+[
(λ x → true) ] process+ f1
(λ _ → lab (getSegm ta seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig ta sig1 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm ta seg1 blocked))
(λ _ → TRAINLEAVE ta seg1 sig1) f0 efq1 f0 efq1
"(taseg1blocked -> TRAINLEAVEtaseg1sig1)"))) f0 efq1 f0 efq1
"(tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1))" ))) f0 efq1 f0 efq1
"(taseg1free -> (tasig1green -> (taseg1blocked -> TRAINLEAVEtaseg1sig1)))
(λ _ → lab (getSegm tb seg1 free))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSig tb sig2 green))
(λ _ → delay (node (process+ f1
(λ _ → lab (setSegm tb seg1 blocked))
(λ _ → TRAINLEAVE tb seg1 sig2) f0 efq1 f0 efq1
"(tbseg1blocked -> TRAINLEAVEtbseg1sig2)" ))) f0 efq1 f0 efq1
"(tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2))" ))) f0 efq1 f0 efq1
"(tbseg1free -> (tbsig2green -> (tbseg1blocked -> TRAINLEAVEtbseg1sig2)))
(λ str str' → strAppend str (strAppend " ||| " str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) Using
(λ str str' → strAppend str (strAppend "[||]" str')) ,
(λ a str → strAppend "fmap (" (strAppend str "))) ,
(λ a str → strAppend "fmap (" (strAppend str "))) (nth [] t))) .[]
(tnode (terc t)) mm' ()

```

badSignal : ¬ (OPTIMIZED-SYSTEM \sqsubseteq_{∞} BADSIGNALS)

badSignal x = noTraceBadSignal nothing badTraceLabels

(x badTraceLabels nothing badTraceBadSignal)

refl refl

A.122 trainExampleCorrectedOptimized3ProofNonRefCutI

```
--@PREFIX@traceCorrectedSystem
```

```
module trainExampleCorrectedOptimized3ProofNonRefinementCutDown where
--module trainExample where
```

```
open import Data.Bool hiding (==)
open import Data.Sum
open import Data.Unit
open import Relation.Binary.PropositionalEquality
open import Data.Bool.Base renaming (T to T') hiding (==)
open import libBool
open import dataAuxFunction
open import process
open import libList
open import Data.String
open import Data.Maybe
open import Data.String renaming (== to ==strb; ++ to ++s)
open import Data.List renaming (++ to ++l; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin
open import Data.Nat hiding (==)
open import parallelSimple
open import interleaved
open import auxData
open import hidingOperator
open import renamingOperator
{- needed possibly for compilation-}
```

```

open import SizedIO.Base
open import renamingResult
open import dataAuxFunction hiding (¬)
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import Data.Empty
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEM ; S
open import trainExampleCorrected
open import TraceWithoutSize
open import RefWithoutSize
-- open import dataAuxFunction
open import choiceSetUOptimized3
open import process2OptimizedProcess3
open import trainExampleCorrectedOptimized3

postulate TODO : {A : Set} → A

--@BEGIN@badTrace
badTraceLabels : List (Label labelTrains)
badTraceLabels = lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: []

badTraceBadSignal : Tr∞ {labelTrains} badTraceLabels nothing BADSIGNALS
badTraceBadSignal = tnode (extc (lab (setSigs sig2 green) :: []) nothing (inj1 zero)
    (tnode (extc [] nothing zero
        (tnode empty))))))

noTraceBadSignal : (m : Maybe (ChoiceSet (∅' ×' ∅' ×' ∅' ×' (∅' ×' ∅'))))
    (l : List (Label labelTrains))
    (tr : Tr∞ {labelTrains} l m OPTIMIZED-SYSTEM)
    (mm' : m ≡ nothing)
    (ll' : l ≡ badTraceLabels)
    → ⊥

noTraceBadSignal = -- Proof Omitted in thesis,
    -- available in the CSP-Agda repository

--@HIDE-BEG
    TODO

--@HIDE-END
badSignalProof : ¬ (OPTIMIZED-SYSTEM ⊑∞ BADSIGNALS)

```

```
badSignalProof x = noTraceBadSignal nothing badTraceLabels
  (x badTraceLabels nothing badTraceBadSignal)
  refl refl
--@END
```

A.123 trainExampleOptimized3.agda

```
--@PREFIX@trainExampleOptimizedThree

module trainExampleOptimized3 where

open import Data.Bool hiding (_==_)
open import Data.Sum
open import Data.Bool.Base renaming (T to T') hiding (_==_)
open import libBool
open import libList
open import Data.String
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List renaming (_++_ to _++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin
open import Data.Nat hiding (_==_)
open import parallelSimple
open import interleave
open import hidingOperator
open import renamingOperator
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction hiding (test)
```

```

open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample hiding (SEGCTL1 ; SEGCTL2 ; SEGCTL ; SYSTEMp1 ; SYSTEMp2
                                SYSTEM ; SYSTEMSHIDE ;
                                SYSTEM-SIGONLY ; main ; TRAINENTER ; TRAINLEAVE)

open import trainExample hiding (main)
open import choiceSetUOptimized3
open import process2OptimizedProcess3

```

```

OPTIMIZED-SYSTEM : Process $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
OPTIMIZED-SYSTEM = optimizedProcess $\infty$   $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ ) SYSTEM-SIGONLY

```

```

main : NativeIO Unit
main = compile OPTIMIZED-SYSTEM

```

```

test : Process  $\infty$  ( $\emptyset' \times' \emptyset' \times' \emptyset' \times' (\emptyset' \times' \emptyset')$ )
test = forcep OPTIMIZED-SYSTEM { $\infty$ }

```

```

test1 : Choice
test1 = Ep test

```

```

test2 : Choice
test2 = lp test

```

```

test3 : Choice
test3 = Tp test

```

A.124 trainExampleOptimized3ProofRefinement.agda

```
--@PREFIX@trainExampleOptimizedThreeProofRefinement
```

```
module trainExampleOptimized3ProofRefinement where
```

```

open import Data.Bool hiding ( $\_ \stackrel{?}{=}$   $\_$ )
open import Data.Sum
open import Data.Maybe

```

```

open import Data.Bool.Base renaming (T to T') hiding (=?_)
open import libBool
open import auxData
open import libList
open import Data.String
open import Data.String renaming (==_ to ==strb_; ++_ to ++s_)
open import Data.List renaming (++_ to ++l_ ; map to mapL)
open import labelUniv
open import Size
open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (=?_)
open import parallelSimple
open import choiceSetU
open import interleave
open import hidingOperator
open import renamingOperator
open import primitiveProcess
open import TraceWithoutSize
open import RefWithoutSize
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample
open import trainExampleOptimized3

badSignal : OPTIMIZED-SYSTEM  $\sqsubseteq_{\infty}$  BADSIGNALS
badSignal .[] .nothing (tnode empty) = tnode empty
badSignal .(lab (setSigs sig1 green) :: [])
    .nothing (tnode (extc .[] .nothing (inj1 zero))

```

```

      (tnode empty))) = tnode (intc (lab (setSigs sig1 green) :: []) nothing zero
                                (tnode (extc [] nothing zero
                                          (tnode empty))))))
-- trace is lab (setSigs sig1 green) :: [] ; nothing
badSignal .(lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: [])
  .nothing (tnode (extc .(lab (setSigs sig2 green) :: [])
    .nothing (inj1 zero) (tnode (extc .[] .nothing zero (tnode empty)))))) =
  tnode (intc (lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: []) nothing zero
    (tnode (extc (lab (setSigs sig2 green) :: []) nothing zero
      (tnode (intc (lab (setSigs sig2 green) :: []) nothing zero
        (tnode (extc [] nothing zero
          (tnode empty))))))))))
-- trace is (lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: []) ; not
badSignal .(lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: efq _ :: l1) m
  (tnode (extc .(lab (setSigs sig2 green) :: efq _ :: l1)
    .m (inj1 zero) (tnode (extc .(efq _ :: l1) .m zero (tnode (extc l1 .m () tr))))))
badSignal .(lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: l) m
  (tnode (extc .(lab (setSigs sig2 green) :: l) .m (inj1 zero)
    (tnode (extc l .m zero (tnode (intc l .m () tr))))))
badSignal .(lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: [])
  .(just (efq $\emptyset\oplus\emptyset$  (inj1 (efq _)))) (tnode (extc .(lab (setSigs sig2 green) :: [])
    .(just (efq $\emptyset\oplus\emptyset$  (inj1 (efq _)))) (inj1 zero) (tnode (extc .[] .(just (efq $\emptyset\oplus\emptyset$  (inj1 (efq _))))))
badSignal .(lab (setSigs sig1 green) :: lab (setSigs sig2 green) :: l) m
  (tnode (extc .(lab (setSigs sig2 green) :: l) .m (inj1 zero) (tnode (extc l .m (suc ())
badSignal .(lab (setSigs sig1 green) :: l1) m (tnode (extc l1 .m (inj1 zero) (tnode (intc l1 .m ()
badSignal .(lab (setSigs sig1 green) :: []) .(just (efq $\emptyset\oplus\emptyset$  (inj1 (efq _))))
  (tnode (extc .[] .(just (efq $\emptyset\oplus\emptyset$  (inj1 (efq _)))) (inj1 zero) (tnode (terc ())))))
badSignal .(lab (setSigs sig1 green) :: l1) m (tnode (extc l1 .m (inj1 (suc ())) tr))
badSignal .(lab (setSigs sig2 green) :: []) .nothing (tnode (extc .[] .nothing (inj2 zero) (tnode em
  tnode (intc (lab (setSigs sig2 green) :: []) nothing (suc zero)
    (tnode (extc [] nothing zero
      (tnode empty))))))
-- trace is lab (setSigs sig2 green) :: [] ; nothing
badSignal .(lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: []) .nothing
  (tnode (extc .(lab (setSigs sig1 green) :: []) .nothing (inj2 zero) (tnode (extc .[] .not
  tnode (intc (lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: []) nothing (suc zero)
    (tnode (extc (lab (setSigs sig1 green) :: []) nothing zero
      (tnode (intc (lab (setSigs sig1 green) :: []) nothing zero
        (tnode (extc [] nothing zero
          (tnode empty))))))))))
-- trace is

```

```

-- lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: [] ; nothing
badSignal .(lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: efq _ :: l1) m
  (tnode (extc .(lab (setSigs sig1 green) :: efq _ :: l1) .m (inj2 zero)
    (tnode (extc .(efq _ :: l1) .m zero (tnode (extc l1 .m () tr))))))
badSignal .(lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: l) m
  (tnode (extc .(lab (setSigs sig1 green) :: l) .m (inj2 zero) (tnode (extc l .m zero (tnode (intc
badSignal .(lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: []))
  .(just (efq0⊕0 (inj2 (efq _)))) (tnode (extc .(lab (setSigs sig1 green) :: []))
  .(just (efq0⊕0 (inj2 (efq _)))) (inj2 zero) (tnode (extc .[] .(just (efq0⊕0 (inj2 (efq _)))) zero (t
badSignal .(lab (setSigs sig2 green) :: lab (setSigs sig1 green) :: l) m
  (tnode (extc .(lab (setSigs sig1 green) :: l) .m (inj2 zero) (tnode (extc l .m (suc ()) tr))))
badSignal .(lab (setSigs sig2 green) :: l1) m (tnode (extc l1 .m (inj2 zero) (tnode (intc .l1 .m () tr))))
badSignal .(lab (setSigs sig2 green) :: []) .(just (efq0⊕0 (inj2 (efq _))))
  (tnode (extc .[] .(just (efq0⊕0 (inj2 (efq _)))) (inj2 zero) (tnode (terc ())))))
badSignal .(lab (setSigs sig2 green) :: l1) m (tnode (extc l1 .m (inj2 (suc ())) tr))
badSignal l m (tnode (intc .l .m (inj1 ())) tr))
badSignal l m (tnode (intc .l .m (inj2 ())) tr))
badSignal .[] .(just (efq0⊕0 (inj1 (efq _)))) (tnode (terc (inj1 ())))
badSignal .[] .(just (efq0⊕0 (inj2 (efq _)))) (tnode (terc (inj2 ())))

```

A.125 trainExampleOptimized3ProofRefCutDownForPaper

```
--@PREFIX@trainExampleOptimizedThreeProofRefinementCutDownForPaper
```

```
module trainExampleOptimized3ProofRefinementCutDownForPaper where
```

```

open import Data.Bool hiding (_==_)
open import Data.Sum
open import Data.Maybe
open import Data.Bool.Base renaming (T to T') hiding (_==_)
open import libBool
open import auxData
open import libList
open import Data.String
open import Data.String renaming (_==_ to _==strb_; _++_ to _++s_)
open import Data.List renaming (_++_ to _++l_ ; map to mapL)
open import labelUniv
open import Size

```

```

open import process
open import choiceSetU
open import choiceFromList
open import preFix
open import simulator
open import NativeIO
open import SizedIO.Console hiding (main)
open import externalChoice
open import Data.Fin

open import Data.Nat hiding (_≡_)
open import parallelSimple
open import choiceSetU
open import interleave
open import hidingOperator
open import renamingOperator
open import primitiveProcess
open import TraceWithoutSize
open import RefWithoutSize
open import SizedIO.Base
open import renamingResult
open import dataAuxFunction
open import Relation.Nullary.Decidable
open import Data.String
open import UnitModule
open import trainExample
open import trainExampleOptimized3

postulate TODO : {A : Set} → A

--@BEGIN@badSignal
badSignal : OPTIMIZED-SYSTEM  $\sqsubseteq_{\infty}$  BADSIGNALS
badSignal = -- Proof Omitted in thesis,
            -- available in the CSP-Agda repository
--@END
  TODO

```

A.126 UnitModule.agda

```

module UnitModule where

```

```
record Unit : Set where
  constructor unit

{-# COMPILER GHC Unit = data () (() #-}
```
